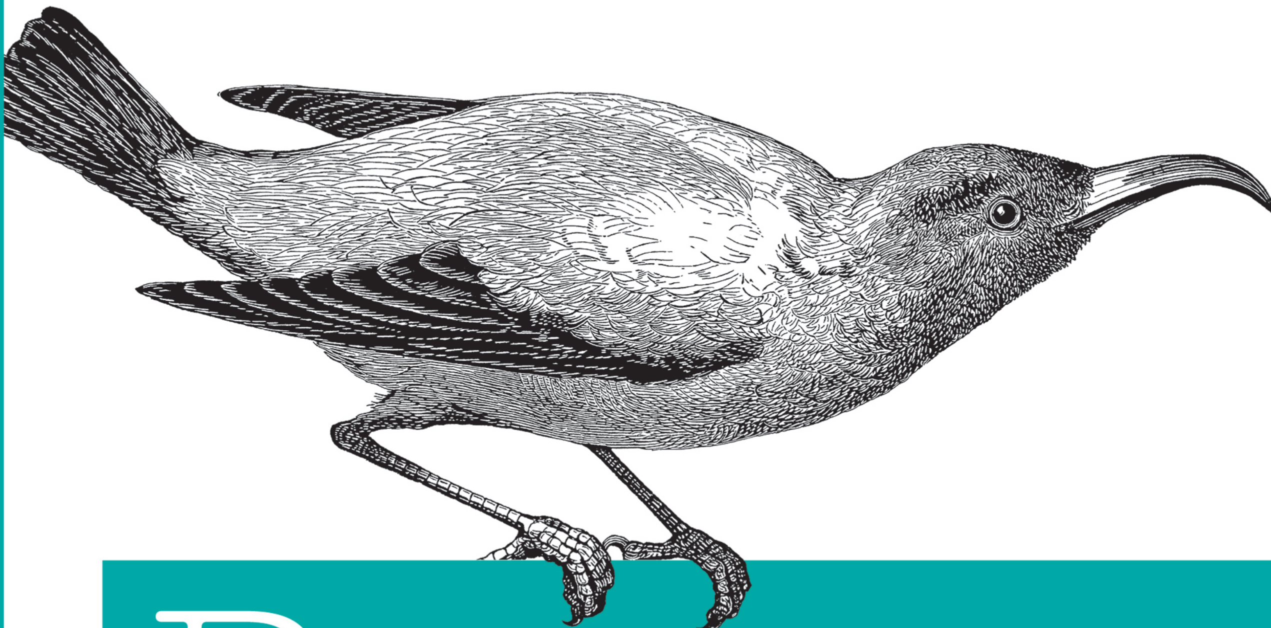
O'REILLY®

TURING 图灵程序设计丛书



# React快速上手开发

React: Up & Running: Building Web Applications

来自Facebook的React入门与开发实践

[保] Stoyan Stefanov 著
张俊达 译

中国工信出版集团 人民邮电出版社
POSTS & TELECOM PRESS

图灵程序设计丛书

# React快速上手开发

## React: Up & Running: Building Web Applications

[保] Stoyan Stefanov 著
张俊达 译

*Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo*
O'Reilly Media, Inc.授权人民邮电出版社出版

人民邮电出版社
北　　京

图书在版编目（C I P）数据

React快速上手开发 / (保加利亚) 斯托扬·斯特凡诺夫 (Stoyan Stefanov) 著 ; 张俊达译. -- 北京 : 人民邮电出版社, 2017.3
（图灵程序设计丛书）
ISBN 978-7-115-44773-9

Ⅰ. ①R… Ⅱ. ①斯… ②张… Ⅲ. ①移动终端—应用程序—程序设计 Ⅳ. ①TN929.53

中国版本图书馆CIP数据核字(2017)第018937号

## 内 容 提 要

本书是 React 入门书。前 3 章介绍如何从空白的 HTML 页面开始构建应用。第 4 章介绍 JSX 语法。从第 5 章开始，你会学习到在实际开发中可能用到的一些附加工具。本书介绍的例子包括 JavaScript 打包工具、单元测试、语法检查、类型、在应用中组织数据流以及不可变数据。

本书适合有 JavaScript 基础的前端开发人员。

◆ 著　　[保] Stoyan Stefanov
译　　张俊达
责任编辑　朱　巍
执行编辑　杨　琳
责任印制　彭志环

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　　印刷

◆ 开本：800×1000　1/16
印张：13
字数：308千字　　2017年3月第1版
印数：1－3 500册　　2017年3月北京第1次印刷

著作权合同登记号　图字：01-2016-10061号

定价：49.00元

**读者服务热线：(010)51095186转600　印装质量热线：(010)81055316**

**反盗版热线：(010)81055315**

**广告经营许可证：京东工商广字第 8052 号**

# O'Reilly Media, Inc.介绍

O'Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O'Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些"细微的信号"来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O'Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O'Reilly 为软件开发人员带来革命性的"动物书"；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 *Make* 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O'Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O'Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版、在线服务或者面授课程，每一项 O'Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

## 业界评论

"O'Reilly Radar 博客有口皆碑。"

——*Wired*

"O'Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。"

——*Business 2.0*

"O'Reilly Conference 是聚集关键思想领袖的绝对典范。"

——*CRN*

"一本 O'Reilly 的书就代表一个有用、有前途、需要学习的主题。"

——*Irish Times*

"Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野，并且切实地按照 Yogi Berra 的建议去做了：'如果你在路上遇到岔路口，走小路（岔路）。'回顾过去，Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。"

——*Linux Journal*

# 目录

# 译者序

这是一本关于 React 的入门书。

React 在近年来广受关注，其声明式 UI、组件化的思想让开发和维护具有复杂交互性的界面变得更容易。目前，前端社区中不断涌现出各种构建 React 应用的方案，各种前端技术会议也多次提及与 React 相关的主题。可以说，React 生态圈正朝着越来越好的方向发展。在翻译本书时，我和我的开发团队正在使用 React、Redux 和 Webpack 作为主要的前端技术栈。我们最直观的感受就是开发过程非常愉悦，思路也更为清晰。

互联网上已经有不少关于 React 的学习资料，但是本书的两个特点让其脱颖而出。一是本书作者 Stoyan 就职于 Facebook，对自家开发的技术自然了解得更加透彻；二是本书把讨论重点放在 React 本身，避免初学者因为 React 相关技术栈过于庞大而望而却步。有一种常见的误解是：你需要花费大量时间在配置工具上，然后才能开始学习 React；但其实真相并非如此。在本书中，我们将从零开始构建一个 React 应用。读者可以在掌握 React 基础后，进行下一步的学习。

感谢稀土掘金的江昇老师引荐，让我得到翻译本书的机会。感谢迅雷云存储前端团队在日常工作和学习中给予我的支持。感谢以下好友阅读本书不同版本的译稿并提出宝贵意见：徐沛文、陈超超、石鑫、黄碧龙。当然还要感谢图灵公司各位编辑老师在本书翻译过程中给予的支持与帮助。

还要特别感谢团子与亭子。

总而言之，这本书的顺利出版离不开大家的努力和帮助。由于水平有限，文中纰漏在所难免，恳请广大读者批评指正。如果对于本书有任何意见或想法，欢迎发送邮件至 bbtzjd@gmail.com。

张俊达<br>2016 年 11 月于深圳

# 前言

大约 2000 年，洛杉矶。这又是一个温暖舒适的加利福尼亚之夜，淡淡的海风轻轻拂来，舒爽惬意。我正准备使用 FTP 把我新建的站点 CSSsprites.com 传送到服务器并向全世界发布。在发布的前几个晚上，我一直在思考一个问题："到底为什么只把 20% 的工作量放在解决应用的主要问题上，却把剩下的 80% 花费在努力克服用户界面的问题上呢？"如果能把所有调用 `getElementById()` 和考虑应用状态（用户上传是否完成？如果上传出错，上传对话框是否要继续显示？）的时间节约出来，我能利用这部分时间完成多少其他的工具呢？为什么界面开发这么耗时？如何处理不同浏览器之间的差异？想到这些，我的大好心情荡然无存。

时间快进到 2015 年 3 月。在当时召开的 Facebook F8 开发者大会上，我所在的团队准备公布两个完全重写的 Web 应用：一个第三方评论模块和一个配套的评论审核工具。和我的小应用 CSSsprites.com 相比，这两个应用非常成熟，功能也复杂得多，并且流量非常大。虽然如此，其开发过程依然令人愉悦。团队中的新成员（甚至包括刚接触 JavaScript 和 CSS 的新手）都能很快地融入其中，轻松高效地贡献功能特性并改进现有代码。团队中的一个成员说："现在我发现这就是自己热爱的一切！"

在这段时间里发生了什么？答案是：React 诞生了。

React 是一个 UI 库，让你只需定义一次用户界面，就可以将其用在多个地方。之后，当应用的状态（state）发生变化时，React 将会自动作出*反应*、更新界面，你无需做其他任何工作。毕竟你已经定义了用户界面。尽管说是定义，其实代码更加偏向*声明式*，你可以使用可管理的小型*组件*构造出一个强大的应用。你再也不需要在函数里花费一半的代码量寻找 DOM 节点了，而是可以只维护应用的状态（通过常规的 JavaScript 对象），把剩下的工作都交给 React 帮你完成。

学习 React 非常划算。一旦学会这个库，便可以使用它构建以下类型的应用：

- Web 应用
- 原生 iOS 和 Android 应用
- Canvas 应用
- TV 应用
- 原生桌面应用

你可以使用与构造组件和用户界面相同的思路，创建具有原生应用性能和控制能力的原生应用（*真正*的原生控制，而不仅仅是看起来像原生）。这并不是指“一次编写，到处运行”（我们的技术尚未实现这一点），而是“一次学习，到处使用”。

简而言之，学习 React 可以帮你节省 80% 的时间，使你可以把精力集中在主要问题上（比如你的应用存在的真正目的）。

# 关于本书

本书从 Web 开发的角度介绍如何学习 React。在前 3 章，你将从一个空白的 HTML 页面开始构建应用。这使得你可以将关注点放在 React 本身，无需了解任何新语法或者辅助工具。

第 4 章介绍 JSX。这是一项单独、可选的技术，通常会同 React 一起使用。

从第 5 章开始，你将学习在实际开发中可能用到的一些附加工具。介绍的例子包括 JavaScript 打包工具（Browserify）、单元测试（Jest）、语法检查（ESLint）、类型（Flow）、在应用中组织数据流（Flux）以及不可变数据（Immutable.js）。所有关于这些辅助技术的讨论都会力求简化，让你依然将精力放在 React 上。你会很快熟悉这些工具的使用，并能根据具体情况选择使用哪些工具。

祝你在学习 React 的过程中一切顺利，大有收获！

# 排版约定

本书使用下列排版约定。

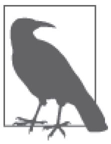
- 等宽字体（`Constant width`）
  表示广义上的计算机编码，包括变量或函数名、数据库、数据类型、环境变量、语句和关键字。

- 等宽粗体（**`Constant width bold`**）
  表示应该由用户按照字面输入的命令或其他文本。

- 等宽斜体（*`Constant width italic`*）
  表示应该由用户替换或取决于上下文的值。



该图标表示提示或建议。



该图标表示一般说明。



该图标表示警告或提醒。

# 代码示例

补充材料（包括代码示例、练习题等）可以从 https://github.com/stoyan/reactbook 下载。

本书旨在帮助你做好工作。一般来说，你可以在程序和文档中使用本书的代码。除非你使用了很大一部分代码，否则无需联系我们获取许可。例如，使用来自本书的几段代码编写一个程序不需要许可。销售和分发 O'Reilly 书中用例的光盘需要许可。通过引用本书用例和代码来回答问题不需要许可。把本书中的大量用例代码并入你的产品文档需要许可。

我们很希望但不强求注明信息来源。一条信息来源通常包括书名、作者、出版社和 ISBN。例如："*React: Up & Running* by Stoyan Stefanov (O'Reilly). Copyright 2016 Stoyan Stefanov, 978-1-491-93182-0"。

如果你感到对示例代码的使用超出了正当引用或者这里给出的许可范围，请随时通过 permissions@oreilly.com 联系我们。

# Safari® 在线图书

Safari Books Online（http://www.safaribooksonline.com）是应运而生的数字图书馆。它同时以图书和视频的形式出版世界顶级技术和商务作家的专业作品。技术专家、软件开发人员、Web 设计师、商务人士和创意专家等，在开展调研、解决问题、学习和认证培训时，都将 Safari Books Online 视作获取资料的首选渠道。

对于组织团体、政府机构和个人，Safari Books Online 提供各种产品组合和灵活的定

价策略。用户可通过一个功能完备的数据库检索系统访问 O'Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 以及其他几十家出版社的上千种图书、培训视频和正式出版之前的书稿。要了解 Safari Books Online 的更多信息，我们网上见。

# 联系我们

请把对本书的评价和问题发给出版社。

美国：

> O'Reilly Media, Inc.
> 1005 Gravenstein Highway North
> Sebastopol, CA 95472

中国：

> 北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
> 奥莱利技术咨询（北京）有限公司

O'Reilly 的每一本书都有专属网页，你可以在那里找到本书的相关信息，包括勘误表、示例以及其他信息。本书的网站地址是：

> http://shop.oreilly.com/product/0636920042266.do

对于本书的评论和技术性问题，请发送电子邮件到：

> bookquestions@oreilly.com

要了解更多 O'Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：

> http://www.oreilly.com

我们在 Facebook 的地址如下：

> http://facebook.com/oreilly

请关注我们的 Twitter 动态：

> http://twitter.com/oreillymedia

我们的 YouTube 视频地址如下：

> http://www.youtube.com/oreillymedia

## 致谢

感谢在本书组稿的不同阶段阅读本书并给我发送反馈和修正意见的所有人：Andreea Manole、Iliyan Peychev、Kostadin Ilov、Mark Duppenthaler、Stephan Alber 和 Asen Bozhilov。

感谢 Facebook 每天开发或者使用 React 以及回答我问题的所有同事。此外还要感谢为 React 提供工具和库以及撰写文章和使用说明的 React 开发者社区。

非常感谢 Jordan Walke。

感谢 O'Reilly 的所有人，本书的出版离不开你们的努力：Meg Foley、Kim Cofer、Nicole Shelby 等。

感谢 Yavor Vatchkov 为本书中的示例应用设计了用户界面（可以在 whinepad.com 体验）。

第 1 章

# Hello World

让我们踏上使用 React 开发应用的旅程吧。在这一章里，你将学习如何设置 React 并编写你的第一个 Hello World 应用。

## 1.1 设置

首先需要获取一份 React 库的源代码。所幸的是，这个过程非常简单。

访问 http://reactjs.com（这个网站会重定向到 React 的官方 GitHub 页面，即 http://facebook.github.io/react/），然后点击 Download 按钮，再点击 Download Starter Kit，即可获得一份 ZIP 文件。解压该文件，并把压缩包中的目录复制到一个方便找到的地方。

例如：

```
mkdir ~/reactbook
mv ~/Downloads/react-0.14.7/ ~/reactbook/react
```

现在你的工作目录（reactbook）应该如图 1-1 所示。

现在，我们只使用其中的 ~/reactbook/react/build/react.js 文件。我们会在后续的学习中逐步介绍其他文件的使用。

需要注意的是，React 并没有强制规定任何目录结构。因此，你可以根据具体情况把 React 移动到其他目录，或者对 react.js 文件进行重命名操作。

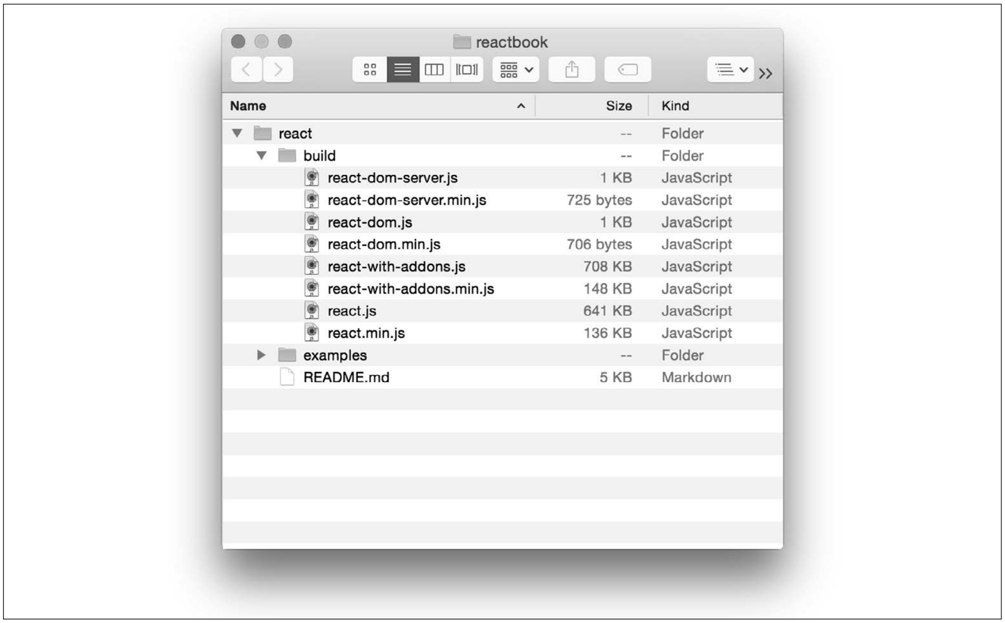



图 1-1：你的 React 目录列表

## 1.2 Hello React World

我们首先在工作目录中编写一个简单的页面（~/reactbook/01.01.hello.html）：

```
<!DOCTYPE html>
<html>
  <head>
    <title>Hello React</title>
    <meta charset="utf-8">
  </head>
  <body>
    <div id="app">
      <!--应用渲染的位置 -->
    </div>
    <script src="react/build/react.js"></script>
    <script src="react/build/react-dom.js"></script>
    <script>
      // 应用的JavaScript代码
    </script>
  </body>
</html>
```

你可以在附带的代码库（https://github.com/stoyan/reactbook/）中找到本书使用的所有代码。

该文件仅有两个值得我们注意的地方：

- 引入了 React 库及其 DOM 插件库（通过 `<script src>` 标签）；
- 定义了这个应用应该出现在页面上的位置（(`<div id="app">`)）。

既可以在 React 应用中混用常规 HTML 内容以及其他 JavaScript 库，也可以在一个页面内使用多个 React 应用。只需要在 DOM 结构中给 React 指定渲染内容的位置即可。

现在我们添加一段输出 Hello world! 的代码。修改 01.01.hello.html 文件，把 JavaScript 注释替换为如下代码：

```
ReactDOM.render(
  React.DOM.h1(null, "Hello world!"),
  document.getElementById("app")
);
```

在浏览器中打开 01.01.hello.html 后，可以看到新的代码在应用中生效了（如图 1-2 所示）。

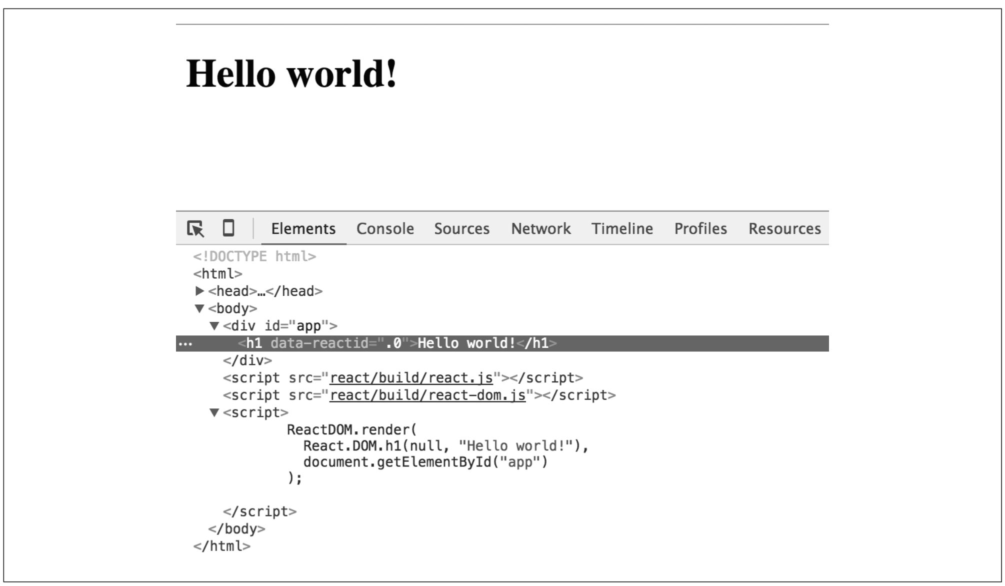



图 1-2：修改后的 Hello World 应用

恭喜，现在你完成了第一个 React 应用！

图 1-2 中的 Chrome 开发者工具显示了 React 生成的代码结构。从中可以看到，`<div id="app">` 中的占位符被 React 应用生成的新内容所代替。

## 1.3 刚才发生了什么

你的第一个应用之所以成功运行，是因为在代码内部发生了一些有趣的事情。

首先，我们使用了 React 对象，所有可用的 API 都可通过该对象进行调用。实际上，React 在设计 API 时注重简化，所以需要记忆的方法名称并不多。

然后，我们使用了 ReactDOM 对象。这个对象只包含几个方法，其中 render() 方法是最有用的。在旧版本中，这些方法曾经属于 React 对象；但从 0.14 版本开始，它们被分离出来，目的是强调应用渲染实际上属于单独的概念。这是因为，你创建的 React 应用还可以渲染到不同的环境中，比如 HTML（浏览器 DOM）、canvas、原生的 Android 或 iOS 应用。

接下来，我们需要关注*组件*的概念。组件可以用于构建用户界面，并通过任何适当的方式进行组合。在实际应用中，你需要创建自定义组件，但在起步阶段，我们先学习使用 React 提供的一个包裹层，它用于包裹 HTML DOM 元素。该包裹层可通过 React.DOM 对象进行调用。在第一个例子中，我们使用了 h1 组件。它对应于 HTML 的 <h1> 元素，可以使用 React.DOM.h1() 方法进行调用。

最后，我们调用了熟悉的 document.getElementById("app") 方法访问 DOM 节点。函数调用通过该参数告诉 React 需要把应用渲染在页面的哪个部分。因此，这是连接你所熟知的 DOM 操作到 React 新大陆的一座桥梁。

> 一旦跨过了从 DOM 到 React 的桥梁，你就不需要再进行 DOM 操作了，因为 React 实现了从组件到基础平台（浏览器 DOM、canvas、原生应用）的转化。你无需再关心 DOM，但这并不意味着不能操作 DOM。React 为开发者提供了选择的自由，你可以根据具体情况，随时回到 DOM 操作的怀抱当中。

理解了每行代码的作用之后，让我们回顾整段代码。刚才发生的事情是：在你选择的 DOM 节点中渲染了一个 React 组件。渲染过程从一个顶层组件开始，而顶层组件可以按需包含许多子元素（子元素中还可以嵌套子元素）。实际上，尽管这是一个简单的例子，但 h1 组件中仍然包含一个子元素，即文本 Hello world!。

## 1.4 React.DOM.*

现在，你知道可以通过 React.DOM 对象把各种各样的 HTML 元素当作 React 组件使用。（图 1-3 展示了如何通过浏览器控制台获取 React.DOM 对象的完整属性列表。）接下来我们将深入了解这个 API。

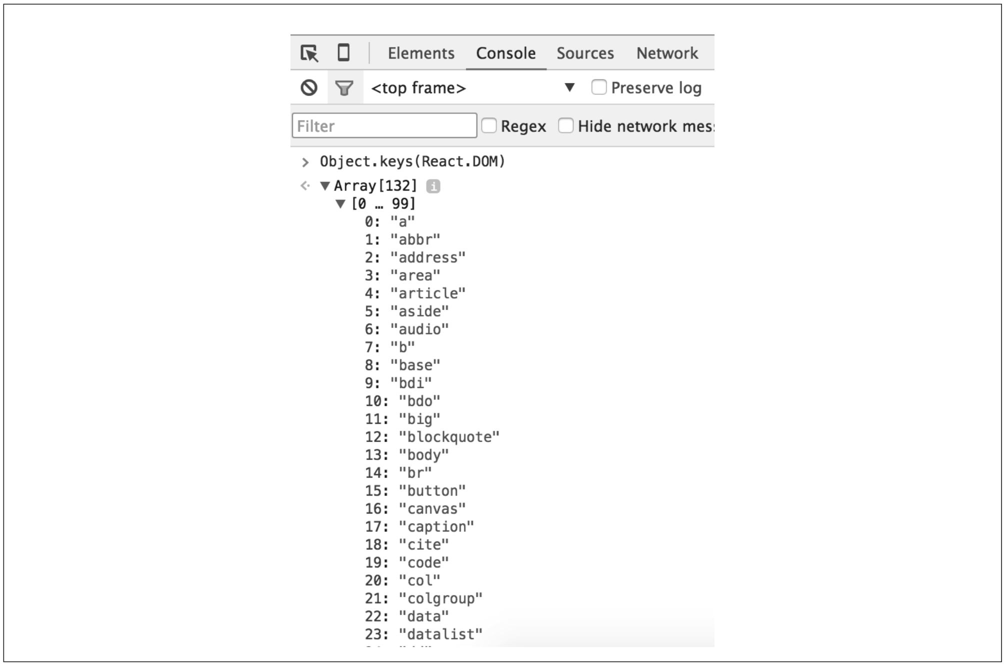



图 1-3：React.DOM 的属性列表

注意 React.DOM 和 ReactDOM 的区别。前者是预定义好的 HTML 元素集合，而后者是在浏览器中渲染应用的一种途径（参考 ReactDOM.render() 方法）。

我们首先看看 React.DOM.* 方法接收的参数。回顾上述 Hello World 应用的代码：

```
ReactDOM.render(
  React.DOM.h1(null, "Hello world!"),
  document.getElementById("app")
);
```

h1() 方法的首个参数接收一个对象（在这个例子中是空对象 null），用于指定该组件的任何属性（比如 DOM 属性）。例如给组件传递 id 属性：

```
React.DOM.h1(
  {
    id: "my-heading",
  },
  "Hello world!"
),
```

上述例子生成的 HTML 结构如图 1-4 所示。

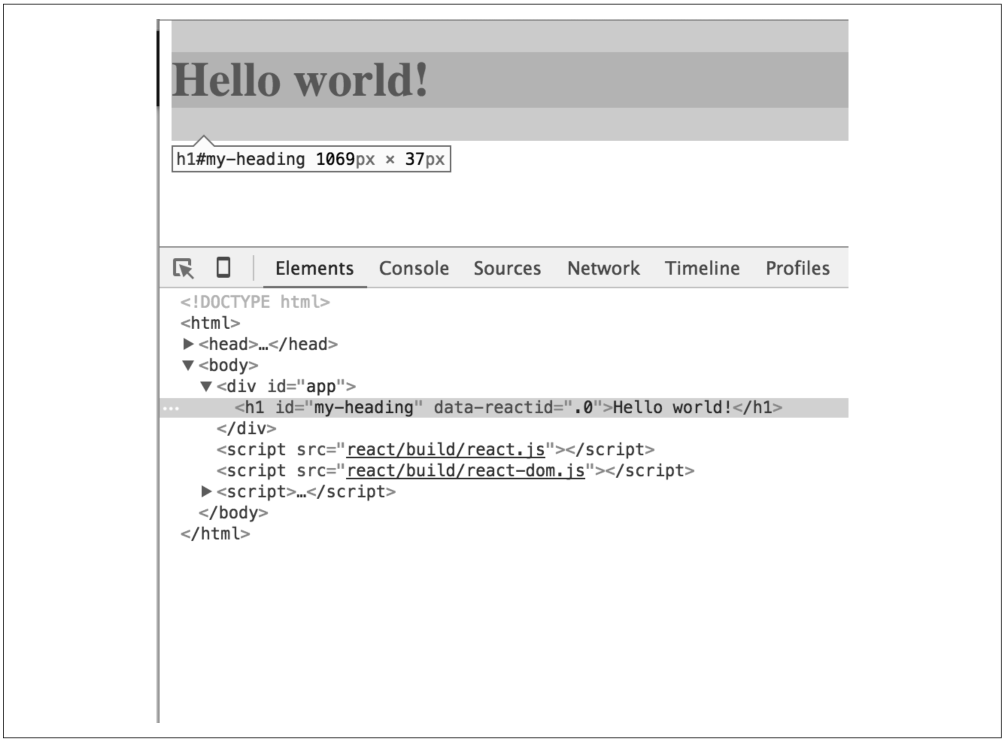



图 1-4：调用 React.DOM 生成的 HTML 代码

第二个参数（在这个例子中是 "Hello world!"）定义了该组件的子元素。最简单的子元素就是上述例子中的文本（在 DOM 结构中是一个文本节点）。此外，你还可以通过传递更多的函数参数，进行子元素的组合与嵌套。比如：

```
React.DOM.h1(
  {id: "my-heading"},
  React.DOM.span(null, "Hello"),
  " world!"
),
```

再看另一个例子，这一次调用了多重嵌套的组件（结果如图 1-5 所示）：

```
React.DOM.h1(
  {id: "my-heading"},
  React.DOM.span(null,
    React.DOM.em(null, "Hell"),
    "o"
  ),
  " world!"
),
```

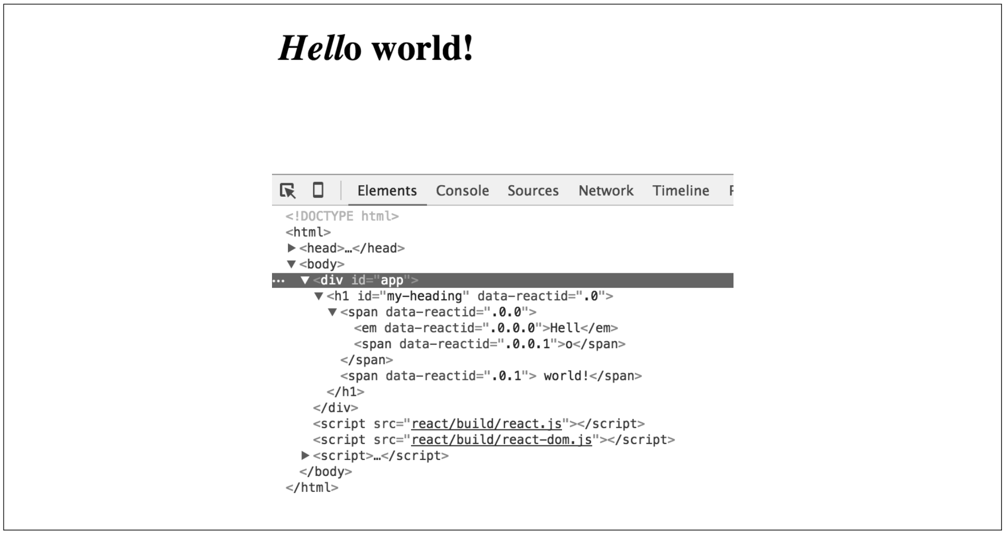



图 1-5：`React.DOM` 嵌套调用生成的 HTML 结构

> 如你所见，当你开始嵌套组件时，使用上述写法会很快遇到大量的函数调用和圆括号。为了简化工作，你可以使用 JSX 语法。我们会在第 4 章将 JSX 作为独立的主题进行探讨；在此之前，我们暂且使用这种纯 JavaScript 语法。原因在于 JSX 会引起一些争议：人们在第一次接触 JSX 时通常会感到排斥（啊，要在 JavaScript 中插入 XML！），但随后就会发现离不开它了。为了让你初步感受一下，这里提供上述代码的 JSX 版本：

```
ReactDOM.render(
  <h1 id="my-heading">
    <span><em>Hell</em>o</span> world!
  </h1>,
  document.getElementById("app")
);
```

## 1.5 特殊 DOM 属性

下列几个 DOM 属性比较特殊，需要引起注意：`class`、`for` 和 `style`。

`class` 和 `for` 不能直接在 JavaScript 中使用，因为它们都是 JavaScript 中的关键字。取而代之的属性名是 `className` 和 `htmlFor`。

```
// 反例
// 属性不会生效
React.DOM.h1(
  {
    class: "pretty",
```

```
    for: "me",
  },
  "Hello world!"
);

// 正确例子
// 属性生效
React.DOM.h1(
  {
    className: "pretty",
    htmlFor: "me",
  },
  "Hello world!"
);
```

至于 `style` 属性，你不能像以往在 HTML 中那样使用字符串对其赋值，而需要使用 JavaScript 对象取而代之。通过避免使用字符串的方式，可以减少跨站脚本（cross-site scripting，XSS）攻击的威胁，因此这是一个广受欢迎的变化。

```
// 反例
// 属性不会生效
React.DOM.h1(
  {
    style: "background: black; color: white; font-family: Verdana",
  },
  "Hello world!"
);

// 正确例子
// 属性生效
React.DOM.h1(
  {
    style: {
      background: "black",
      color: "white",
      fontFamily: "Verdana",
    }
  },
  "Hello world!"
);
```

此外，在处理 CSS 属性时，还要注意使用 JavaScript API 的属性名。换句话说，就是使用 `fontFamily` 代替 `font-family`。

## 1.6　React DevTools 浏览器扩展

如果你在尝试本章前面的例子时，打开过浏览器控制台，你会看到一条提示信息"Download the React DevTools for a better development experience: https://fb.me/react-devtools"。这个 URL 是通往浏览器扩展安装页的链接，该扩展可以帮助你调试 React 应用（如图 1-6 所示）。

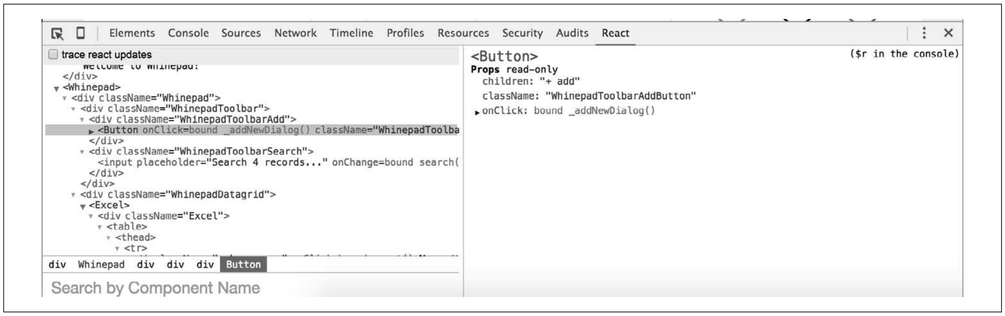



图 1-6：浏览器扩展程序 React DevTools

虽然这个浏览器扩展在开始阶段可能派不上用场，但到了第 4 章你就会发现它的意义所在。

## 1.7　下一步：自定义组件

至此，你已经完成了 Hello World 应用的骨架。现在你知道如何：

- 安装、设置并使用 React 库（仅仅需要引入两个 `<script>` 标签）；
- 在选定的 DOM 节点中渲染一个 React 组件（比如 `ReactDOM.render(reactWhat, domWhere)`）；
- 使用内建组件，即那些常规 DOM 元素外的包裹层（比如 `React.DOM.div(attributes, children)`）。

然而，React 真正的力量要在你开始使用自定义组件构建（并更新）应用界面后才能体现出来。下一章，我们将学习组件的具体操作。

# 第 2 章

# 组件的生命周期

现在你已经知道如何使用预定义的 DOM 组件，是时候学习如何建立自己的组件了。

## 2.1　基础

创建新组件的 API 如下：

```
var MyComponent = React.createClass({
  /* 组件详细说明 */
});
```

在上述例子中，“组件详细说明”是一个 JavaScript 对象，该对象需要包含一个名为 `render()` 的方法以及一系列可选的方法与属性。一个基本的例子大概如下所示：

```
var Component = React.createClass({
  render: function() {
    return React.DOM.span(null, "I'm so custom");
  }
});
```

如你所见，唯一必须要做的事情就是实现 `render()` 方法。该方法必须返回一个 React 组件，不能只返回文本内容。这也是上述代码片段中使用 `span` 组件的原因。

在应用中，使用自定义组件的方法和使用 DOM 组件的方法类似：

```
ReactDOM.render(
  React.createElement(Component),
  document.getElementById("app")
);
```

该自定义组件的渲染结果如图 2-1 所示。




图 2-1：你的第一个自定义组件

`React.createElement()` 是创建组件“实例”的方法之一。如果你想创建多个实例，还有另一种途径，就是使用工厂方法：

```
var ComponentFactory = React.createFactory(Component);

ReactDOM.render(
  ComponentFactory(),
  document.getElementById("app")
);
```

请注意，我们之前介绍的 `React.DOM.*` 方法实际上只是在 `React.createElement()` 的基础上进行了一层封装。换句话说，以下代码同样可以渲染 DOM 组件：

```
ReactDOM.render(
  React.createElement("span", null, "Hello"),
  document.getElementById("app")
);
```

如你所见，和自定义组件使用 JavaScript 函数进行定义不同，DOM 元素是使用字符串定义的。

## 2.2 属性

你的组件可以接收属性，并根据属性值进行相对应的渲染或表现。所有属性都可以通过 this.props 对象获取。让我们看看下面这个例子：

```
var Component = React.createClass({
  render: function() {
    return React.DOM.span(null, "My name is " + this.props.name);
  }
});
```

在渲染组件时，传递属性的方法如下：

```
ReactDOM.render(
  React.createElement(Component, {
    name: "Bob",
  }),
  document.getElementById("app")
);
```

结果如图 2-2 所示。




图 2-2：使用组件属性

请把 this.props 视作只读属性。从父组件传递配置到子组件时，属性非常重要。（从子组件到父组件也是这样，你会在本书的随后章节中了解。）如果你想为 this.props 设置属性，只需要使用额外的变量或者组件详细说明对象的属性即可（比如 this.thing 对应于 this.props.thing）。事实上，在支持 ECMAScript5 的浏览器中，你不能改变 this.props，因为：

```
> Object.isFrozen(this.props) === true; // true
```

## 2.3 propTypes

在你的组件中，可以添加一个名为 propTypes 的属性，以声明组件需要接收的属性列表及其对应类型。下面是一个例子：

```
var Component = React.createClass({
  propTypes: {
    name: React.PropTypes.string.isRequired,
  },
  render: function() {
    return React.DOM.span(null, "My name is " + this.props.name);
  }
});
```

虽然也可以不使用 propTypes，但是使用 propTypes 有以下两方面的好处。

- 通过预先声明组件期望接收的参数，让使用组件的用户不需要在 render() 方法的源代码中到处寻找该组件可配置的属性（这可能需要花费很长时间）。
- React 会在运行时验证属性值的有效性。这使得你可以放心编写 render() 函数，而不需要对组件接收的数据类型有所顾虑（甚至过分怀疑）。

让我们看看其验证效果。name: React.PropTypes.string.isRequired 清晰地指明了 name 属性是一个必须提供的字符串值。假设你忘记传递这个值，会在控制台中得到一个警告信息（如图 2-3 所示）：

```
ReactDOM.render(
  React.createElement(Component, {
    // name: "Bob",
  }),
  document.getElementById("app")
);
```

如果你提供了一个无效类型的值，当然也会得到警告信息。比如提供的值是整数类型（如图 2-4 所示）：

```
React.createElement(Component, {
  name: 123,
})
```




图 2-3：没有提供必需的属性值时出现的警告信息

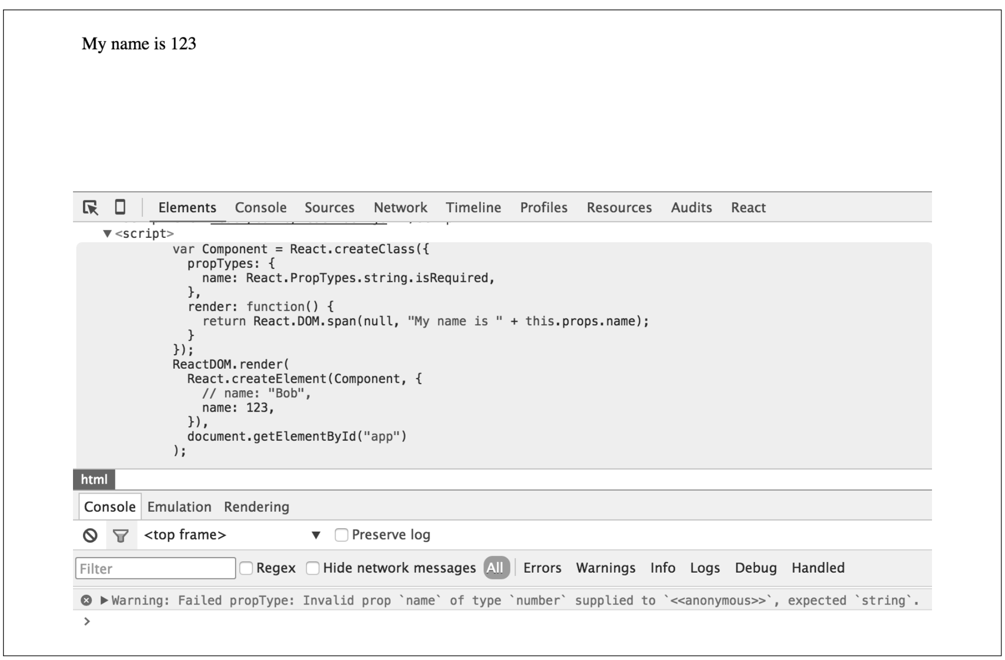



图 2-4：提供了无效类型的属性值时出现的警告信息

图 2-5 列出了 PropTypes 的可用属性列表，用于根据具体需求进行选用。

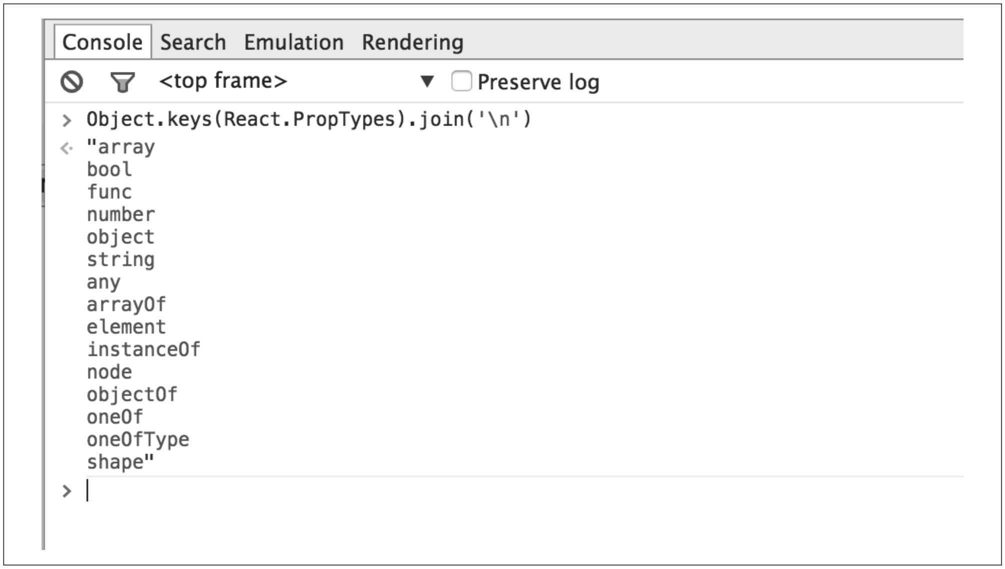



图 2-5：所有的 React.PropTypes 可用属性

> 在组件中声明 propTypes 是可选的，这也意味着你可以在这里选择列出部分而非所有属性。或许你会觉得不声明所有属性是一个坏主意，但要牢记一点，这种情况在调试他人代码时是有可能遇到的。

## 默认属性值

当你的组件接收可选参数时，你需要特别小心没有提供这些属性的情况，以确保组件在这些情况下正常运行。这难免会导致一些防御性的样板代码产生，比如：

```
var text = 'text' in this.props ? this.props.text : '';
```

可以通过实现 getDefaultProps() 方法，避免这种代码的产生（并把关注点放在更重要的地方）：

```
var Component = React.createClass({
  propTypes: {
    firstName: React.PropTypes.string.isRequired,
    middleName: React.PropTypes.string,
    familyName: React.PropTypes.string.isRequired,
    address: React.PropTypes.string,
  },

  getDefaultProps: function() {
```

```
        return {
          middleName: '',
          address: 'n/a',
        };
      },

      render: function() {/* ... */}
    });
```

如你所见，getDefaultProps() 方法返回一个对象，并为每个可选属性（不带 .isRequired 的属性）提供了默认值。

## 2.4　state

到目前为止，我们的例子都是纯静态的（或者说是“无状态”的），旨在给你一种使用组件块组合界面的思路。不过 React 真正的闪光点出现在应用数据发生改变的时候（也是传统的浏览器 DOM 操作和维护变得复杂的地方）。React 有一个称为 state 的概念，也就是组件渲染自身时用到的数据。当 state 发生改变时，React 会自动重建用户界面。因此，当你（在 render() 方法中）初始化构造界面后，只需要关心数据的变化即可。你完全不需要再关心界面变化了。毕竟，你的 render() 方法已经提供了组件的蓝图。

> 调用 setState() 后的界面更新是通过一个队列机制高效地进行批量修改的，直接改变 this.state 会导致意外行为的发生，因此你不应该这么做。和前面的 this.props 类似，可以把 this.state 当作只读属性；否则，不仅仅在语义上不够直观，还会导致不可预料的结果。类似地，永远不要自行调用 this.render() 方法——而是将其留给 React 进行批处理，计算最小的变化数量，并在合适的时机调用 render()。

和 this.props 的取值方式类似，你可以通过 this.state 对象取得 state。在更新 state 时，可以使用 this.setState() 方法。当 this.setState() 被调用时，React 会调用你的 render() 方法并更新界面。

> 当 setState() 被调用时，React 会更新界面。这是最为常见的情形，但你在随后的学习中会了解到一种例外情况。可以通过令一个名为 shouldComponentUpdate() 的特殊“生命周期”方法返回 false，从而避免界面更新。

## 2.5　带状态的文本框组件

下面来构建一个新组件。这是一个可以记录已输入字符数的文本框组件（如图 2-6 所示）。

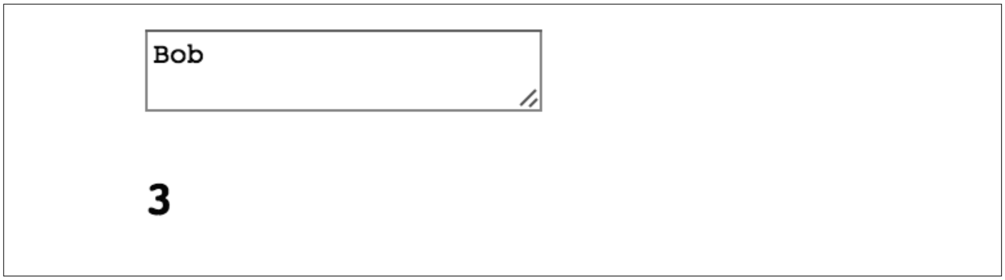



图 2-6：自定义文本框组件的最终效果

你（和其他使用这个可重用组件的人）可以这样使用这个新组件：

```
ReactDOM.render(
  React.createElement(TextAreaCounter, {
    text: "Bob",
  }),
  document.getElementById("app")
);
```

现在让我们一起实现这个组件。我们可以先创建一个类似于上述例子的“无状态”版本，暂不处理状态的更新：

```
var TextAreaCounter = React.createClass({
  propTypes: {
    text: React.PropTypes.string,
  },

  getDefaultProps: function() {
    return {
      text: '',
    };
  },

  render: function() {
    return React.DOM.div(null,
      React.DOM.textarea({
        defaultValue: this.props.text,
      }),
      React.DOM.h3(null, this.props.text.length)
    );
  }
});
```

在上述例子中，你可能注意到文本框使用了 `defaultValue` 属性，而不是你在常规 HTML 中习惯使用的文本子元素。这是因为 React 和传统 HTML 在处理表单上有一些细微的区别。我们会在第 4 章讨论这一点，不过请放心，二者之间没有太多区别。此外，你将会发现这些不同之处的存在是合理的，会让你的开发体验更加愉悦。

如你所见，这个组件接收一个可选字符串属性 text，并使用给定的属性值渲染一个文本框组件，以及一个简单显示字符串长度的 <h3> 组件。

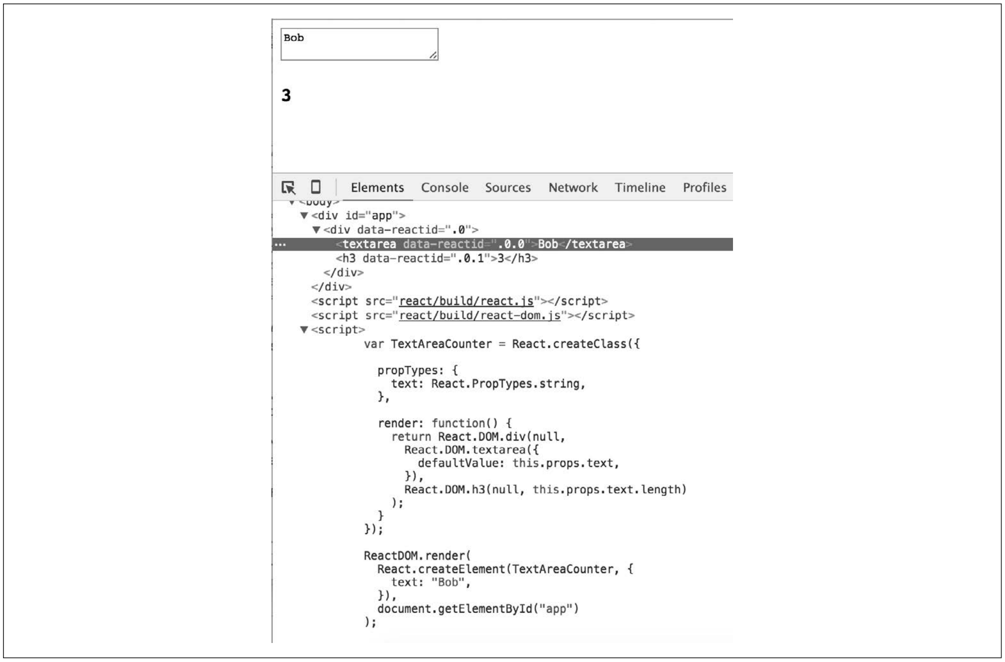



图 2-7：目前的 TextAreaCounter 组件

接下来需要把无状态组件变为带状态的。换句话说，我们要让组件维护某些数据（state）。这些数据在初始化渲染时需要用到，并且在数据随后发生改变时进行更新（重新渲染）。

在你的组件中实现一个 getInitialState() 方法，以保证总是可以合法地取得数据。

```
getInitialState: function() {
  return {
    text: this.props.text,
  };
},
```

这个组件仅仅需要维护 textarea 组件中的文本数据，因此 state 只包含一个属性 text，该属性可以通过 this.state.text 访问。在初始化时（即 getInitialState() 函数执行时），仅仅是把 text 属性复制过来。随后在数据发生改变时（即用户在文本框中输入内容时），组件可以通过一个辅助方法更新 state：

```
_textChange: function(ev) {
  this.setState({
    text: ev.target.value,
  });
},
```

改变 state 必须使用 `this.setState()` 方法。该方法接收一个对象参数，并把对象与 `this.state` 中已存在的数据进行合并。或许你已经猜到，`_textChange()` 就是一个事件监听器，可以接收一个事件对象 `ev` 并通过该参数取得文本框的内容。

最后要做的就是在 `render()` 方法中使用 `this.state` 代替 `this.props`，并设置事件监听器：

```
render: function() {
  return React.DOM.div(null,
    React.DOM.textarea({
      value: this.state.text,
      onChange: this._textChange,
    }),
    React.DOM.h3(null, this.state.text.length)
  );
}
```

现在无论用户何时在文本框中输入数据，计数器的值都会自动反映字符长度的变化（如图 2-8 所示）。

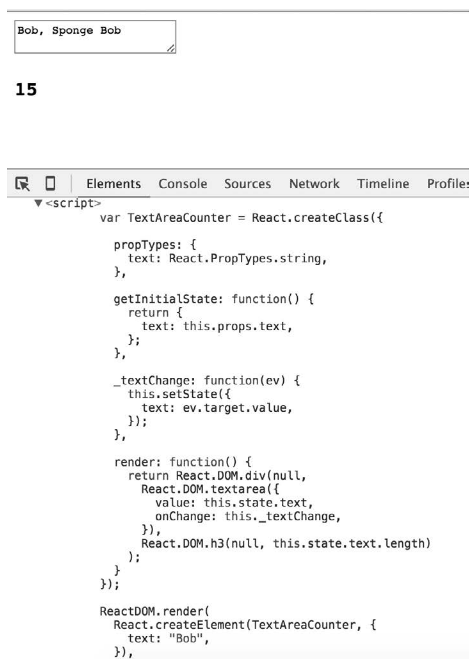



图 2-8：在文本框中输入内容

## 2.6　关于 DOM 事件的说明

为了避免混淆，关于以下这行代码有几点需要说明：

```
onChange: this._textChange
```

出于性能、便捷性与合理性考虑，React 使用了自身的合成事件系统。为了帮助你理解为什么这样做，我们先来看看在原始 DOM 世界里是怎么实现事件处理的。

## 2.6.1 传统的事件处理

使用内联事件处理器是非常方便的，就像这样：

```
<button onclick="doStuff">
```

尽管这样做很方便也易于阅读（事件处理和界面代码放在一起），但是当很多事件处理函数像这样散落在界面代码各处时，就显得效率低下了。而且在同一个按钮中设置多个事件监听器也很困难，特别是当按钮位于其他人的“组件”中时，你是不会愿意到他们的代码中修改源代码的。这就是在 DOM 世界中，人们使用 `element.addEventListener`（这样会导致代码分散到两个甚至多个地方）和*事件委托*（为了解决性能问题）设置事件监听的原因。事件委托意味着你可以在某个父节点监听事件。比如当一个 `<div>` 包含许多按钮时，只需要为所有按钮设置一个监听器即可。

使用事件委托的代码如下所示：

```
<div id="parent">
  <button id="ok">OK</button>
  <button id="cancel">Cancel</button>
</div>

<script>
document.getElementById('parent').addEventListener('click', function(event) {
  var button = event.target;

  // 根据具体点击的按钮进行不同操作
  switch (button.id) {
    case 'ok':
      console.log('OK!');
      break;
    case 'cancel':
      console.log('Cancel');
      break;
    default:
      new Error('Unexpected button ID');
  };
});
</script>
```

尽管方法奏效，性能也不错，但是它仍有一些缺点。

- 监听器的声明代码远离视图组件，使得代码难以搜索与调试。

- 使用委托总要经过 switch 结构，在你开始进行实际逻辑编写之前，需要创建不必要的样板代码。
- 实际中为处理浏览器的不一致性(上述代码中省略了这一步骤),会让代码变得更加冗长。

不幸的是，在你打算把这段代码放到真实环境给用户使用之前，还需要做更多工作，以支持所有主流浏览器。

- 除了 `addEventListener` 之外还需要 `attachEvent`。
- 要在监听器顶部使用 `event = event || window.event;`。
- 需要使用 `var button = event.target || event.srcElement;`。

尽管以上所有都是必需的，但是非常令人厌烦，因此你最终可能会选用某种类型的事件库作为代替。然而，React 自带一套解决方案来帮助你摆脱事件处理的噩梦，为什么还要添加另一个库（并学习更多的 API）呢？

### 2.6.2 React 的事件处理

为了包裹并规范浏览器事件，React 使用了合成事件来消除浏览器之间的不一致情况。有了 React 的帮助，现在你可以依靠 `event.target` 在所有浏览器中取得想要的值了。这就是在 `TextAreaCounter` 代码片段中，你只需要使用 `ev.target.value` 就可以正常工作的原因。与此同时，取消事件的 API 在所有浏览器中都通用了；`event.stopPropagation()` 和 `event.preventDefault()` 甚至在老版本 IE 浏览器中也可以生效。

这种语法轻松地把视图和事件监听绑定在一起。虽然其语法看起来就像传统的内联事件处理器一样，但背后的实现原理并非如此。事实上，React 基于性能考虑，使用了事件委托。

此外，React 在事件处理中使用驼峰法命名，因此你需要使用 `onClick` 代替 `onclick`。

如果你出于某种原因需要使用原生的浏览器事件，可以使用 `event.nativeEvent`，但估计你不太可能会用得上。

还有一件事情需要注意。`onChange` 事件（在文本框例子中已经用到）的行为和你预期中是一样的：当用户输入时触发，而不是像原生 DOM 事件那样，在用户结束输入并把焦点从输入框移走时才触发。

## 2.7 props 与 state

现在你知道，当你需要在 `render()` 方法中显示组件时，可以访问 `this.props` 和 `this.state` 了。或许你会有疑问，两者应分别在何时使用呢？

属性是一种给外部世界设置组件的机制，而状态则负责组件内部数据的维护。因此，如

果与面向对象编程进行类比的话，this.props 就像是传递给类构造函数的参数，而 this.state 则包含了你的私有属性。

## 2.8　在初始化 state 时使用 props：一种反模式

在前面的一个例子中，我们在 getInitialState() 方法中使用了 this.props：

```
getInitialState: function() {
  return {
    text: this.props.text,
  };
},
```

通常认为这种做法是反模式。理想情况下，你可以在 render() 方法中将 this.state 和 this.props 任意组合，以进行界面构建。但有时候，你想要传递一个值到组件中，用于构造初始状态。这种想法本身没什么不对，但如果组件的调用者以为属性（在之前的例子中是 text 属性）总是能保持最新，这种写法就有歧义了。为了符合语境，对命名作一点小改动就足够了。比如，把属性名 text 改成 defaultText 或者 initialValue：

```
propTypes: {
  defaultValue: React.PropTypes.string
},

getInitialState: function() {
  return {
    text: this.props.defaultValue,
  };
},
```

第 4 章将说明 React 如何实现自己的 input 和 textarea 来解决这个问题。这和人们之前所了解的 HTML 知识有一些出入。

## 2.9　从外部访问组件

在实际中，你不会总是从零开始构建一个 React 应用。有时候，你可能需要将一个现有的应用或网站逐步迁移到 React。幸运的是，React 的设计允许它和其他任何已存在的代码共存。毕竟，React 的原作者也并没有从头完全用 React 对一整个超大型应用（Facebook）进行重构。

让你的 React 应用和外界进行通信的一种方法，是在使用 ReactDOM.render() 方法进行渲染时，把引用赋值给一个变量，然后在外部通过该变量访问组件：

```
var myTextAreaCounter = ReactDOM.render(
  React.createElement(TextAreaCounter, {
    defaultValue: "Bob",
  }),
  document.getElementById("app")
);
```

现在你可以通过 `myTextAreaCounter` 访问组件的方法和属性，就像在组件内部使用 `this` 访问一样。你甚至可以在 JavaScript 控制台中操控这个组件（如图 2-9 所示）。




图 2-9：通过引用访问已渲染的组件

以下这行代码设置了新的 state 值：

```
myTextAreaCounter.setState({text: "Hello outside world!"});
```

以下这行代码获取了 React 创建的父元素 DOM 节点的引用：

```
var reactAppNode = ReactDOM.findDOMNode(myTextAreaCounter);
```

获取 DOM 结构中首个 `<div id="app">` 节点。这也是你让 React 进行渲染的位置：

```
reactAppNode.parentNode === document.getElementById('app'); // true
```

通过以下方法可以访问组件的属性和状态：

```
myTextAreaCounter.props; // Object { defaultValue: "Bob"}
myTextAreaCounter.state; // Object { text: "Hello outside world!"}
```

现在你已经有权限从组件外部访问整个组件的 API。尽管如此，你还是应当谨慎使用这种超能力。也许当你需要获取节点的尺寸并确保其适配整个页面时，可以使用 `ReactDOM.findDOMNode()`，但这种情况确实出现得不多。即便你认为这样可以方便地修改那些不属于你的组件的 state，可这样做违背了 React 的初衷。假若组件并不能预料到外部的这些干扰，可能还会导致 bug 的产生。比如以下的代码虽然可以生效，但不推荐你这样做：

```
// 反例
myTextAreaCounter.setState({text: 'NOOOO'});
```

## 2.10 中途改变属性

正如你已经知道的，属性是配置组件的一种方式。因此在组件创建完成后，从外部改变组件的属性也是合理的。但你的组件应当作好应对这种场景的准备。

如果你回去看看之前例子中的 `render()` 方法，会发现我们在该方法中只用到了 `this.state`：

```
render: function() {
  return React.DOM.div(null,
    React.DOM.textarea({
      value: this.state.text,
      onChange: this._textChange,
    }),
    React.DOM.h3(null, this.state.text.length)
  );
}
```

如果你从外部改变了组件属性，不会对渲染产生影响。换句话说，当你进行如下操作时，文本框中的内容不会发生变化：

```
myTextAreaCounter = ReactDOM.render(
  React.createElement(TextAreaCounter, {
    defaultValue: "Hello", // 之前的值为字符串"Bob"
  }),
  document.getElementById("app")
);
```

尽管 `myTextAreaCounter` 会在 `ReactDOM.render()` 重新调用时被覆盖，但是应用的原有状态依然会保留。React 会在应用改变前后作出协调（reconciliation），且不会消除之前的数据。相反，React 仅作出最小限度的修改。

现在 `this.props` 的内容发生了变化（但界面没有变化）：

```
myTextAreaCounter.props; // 对象{defaultValue="Hello"}
```

设置 state 确实会更新界面：

```
// 反例
myTextAreaCounter.setState({text: 'Hello'});
```

但这种做法是不好的，因为这在更复杂的组件中可能会导致不一致的状态；比如打乱内部计数器、布尔型标记、事件监听器等。

如果你希望优雅地处理外部入侵（即属性改变），可以通过 `componentWillReceiveProps()` 方法实现：

```
componentWillReceiveProps: function(newProps) {
  this.setState({
    text: newProps.defaultValue,
  });
},
```

如你所见，这个方法会接收新属性对象，让你可以根据新属性设置 state。此外，还可以进行其他工作以确保组件状态保持正常。

## 2.11 生命周期方法

上述代码片段用到的 `componentWillReceiveProps()` 方法是 React 提供的所谓生命周期方法之一。你可以使用生命周期方法监听组件的改变。除此之外，你可以实现的其他生命周期方法还包括以下几个。

- `componentWillUpdate()`

  当你的组件再次渲染时，在 `render()` 方法前调用（在组件的 `props` 或者 `state` 发生改变时会触发该方法）。

- `componentDidUpdate()`

  在 `render()` 函数执行完毕，且更新的组件已被同步到 DOM 后立即调用。该方法不会在初始化渲染时触发。

- `componentWillMount()`

  在新节点插入 DOM 结构之前触发。

- `componentDidMount()`

  在新节点插入 DOM 结构之后触发。

- `componentWillUnmount()`

  在组件从 DOM 中移除时立刻触发。

- `shouldComponentUpdate(newProps, newState)`

  这个方法在 `componentWillUpdate()` 之前触发，给你一个机会返回 `false` 以取消更新组件，这意味着 `render()` 方法将不会被调用。这在性能关键型的应用场景中非常有用。当你认为变更的内容没什么特别或者没有重新渲染的需要时，可以实现该方法。要决定是否更新，只需比较 `newState` 参数和目前的状态 `this.state` 的区别，以及 `newProps` 参数和目前的属性 `this.props` 的区别。当然，也可直接认为该组件是静态的而无需更新。（随后你会见到相关例子。）

## 2.12　生命周期示例：输出日志记录

为了更好地理解组件的生命周期，我们基于 `TextAreaCounter` 组件添加一些日志记录。我们简单地实现所有的生命周期方法，并且在这些方法被调用时，输出其自身的方法名和参数到控制台中：

```
var TextAreaCounter = React.createClass({

  _log: function(methodName, args) {
    console.log(methodName, args);
  },
  componentWillUpdate: function() {
    this._log('componentWillUpdate', arguments);
  },
  componentDidUpdate: function() {
    this._log('componentDidUpdate', arguments);
  },
  componentWillMount: function() {
    this._log('componentWillMount', arguments);
  },
  componentDidMount: function() {
    this._log('componentDidMount', arguments);
  },
  componentWillUnmount: function() {
    this._log('componentWillUnmount', arguments);
  },

  // ...
  // 其他方法,包括render()等

};
```

图 2-10 显示了页面加载完成后的结果。

如你所见，有两个方法被调用了，但是没有携带任何参数。`componentDidMount()` 方法是两者中较为有意思的一个。如果需要，你可以在这个方法中通过 `ReactDOM.findDOMNode(this)` 方法访问刚加载好的 DOM 节点，比如获取组件元素的尺寸大小。由于此时组件已经渲染，你可以进行各种初始化工作。

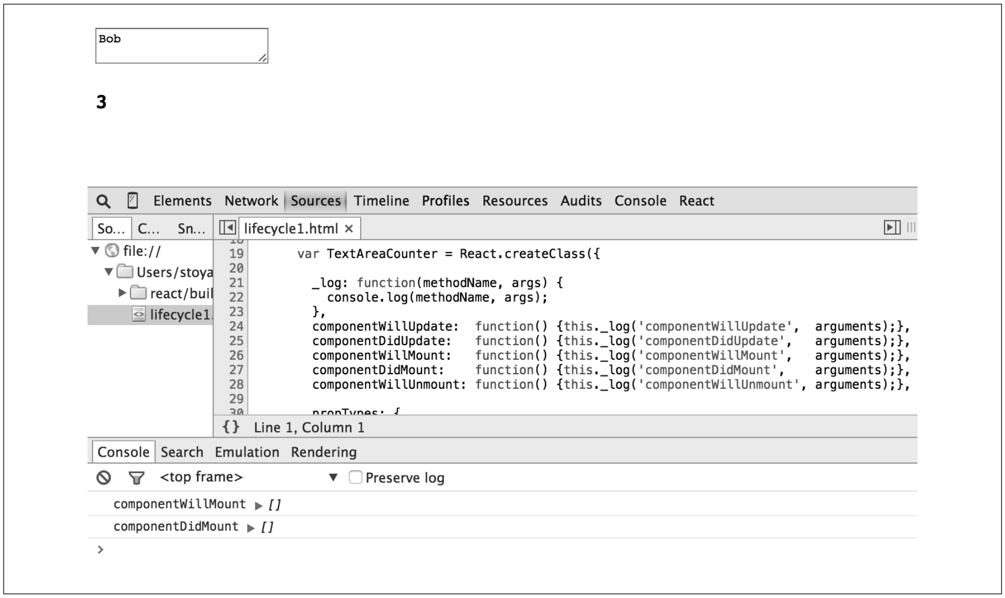



图 2-10：加载组件

接下来，假如你在文本框末尾多输入一个 s，使文本变成 Bobs，会发生什么事情呢？（如图 2-11 所示。）




图 2-11：更新组件

componentWillUpdate(nextProps, nextState) 方法被调用了，新的数据将会被用于重新渲染组件。第一个参数是 this.props 的新值（在这个例子中没有发生变化）。第二个参数是 this.state 的新值。第三个参数是 context，目前我们并不关心这个参数。你可以通过比较参数（比如 newProps）和目前的值 this.props 来决定是否需要进行操作。

在 componentWillUpdate() 方法执行后，componentDidUpdate(oldProps, oldState) 方法被调用，它接收的两个参数分别是原本的 props 和 state。这个方法让你有机会在组件改变之后进行操作。你可以在该方法中执行 this.setState()，但你在 componentWillUpdate() 中可不能这么做。

假设你想要限制文本框中输入的字符数，应该在用户输入时调用的事件处理器 _textChange() 中进行限制。但如果有人（假设是个比你更天真的年轻人）试图从外部调用 setState() 方法呢？（之前我们提到过，这是一种不好的做法。）你能否继续维持组件的正常状态呢？答案是肯定的。你可以在 componentDidUpdate() 方法中验证字符长度是否大于允许值；如果大于，就把 state 恢复到之前的状态。就像这样：

```
componentDidUpdate: function(oldProps, oldState) {
  if (this.state.text.length > 3) {
    this.replaceState(oldState);
  }
},
```

虽然这有点杞人忧天，但确实是一种可行的方案。

> 注意这里使用了 replaceState() 代替 setState()。setState(obj) 会把属性和当前的 this.state 进行合并，而 replaceState() 则会完全重写所有状态。

## 2.13 生命周期示例：使用 mixin

在前面的例子中，五个生命周期方法中的四个已经被日志记录下来。当子组件从父组件中移除时，最能体现第五个方法 componentWillUnmount() 的作用。在下面这个例子中，要把父子组件的所有变化都记录下来。为了实现代码复用，接下来引入一个新概念：mixin。

mixin 其实是一个 JavaScript 对象，包含一系列方法和属性。mixin 不能独立使用，需要包含在另一个对象的属性中。在这个日志例子中，mixin 可以写成这样：

```
var logMixin = {
  _log: function(methodName, args) {
    console.log(this.name + '::' + methodName, args);
  },
  componentWillUpdate: function() {
```

```
    this._log('componentWillUpdate', arguments);
  },
  componentDidUpdate: function() {
    this._log('componentDidUpdate', arguments);
  },
  componentWillMount: function() {
    this._log('componentWillMount', arguments);
  },
  componentDidMount: function() {
    this._log('componentDidMount', arguments);
  },
  componentWillUnmount: function() {
    this._log('componentWillUnmount', arguments);
  },
};
```

在非 React 世界中，你可以使用 `for-in` 循环，把所有属性复制到一个新对象中，并通过这种方法让新对象获得 mixin 的所有功能。在 React 世界中，你可以通过 `mixins` 属性快速实现，就像这样：

```
var MyComponent = React.createClass({

  mixins: [obj1, obj2, obj3],

  // 其他所有方法

};
```

只需要把一个 JavaScript 对象数组赋值给 `mixins` 属性，然后把剩下的工作交给 React。把 `logMixin` 包含到组件中的方法如下：

```
var TextAreaCounter = React.createClass({
  name: 'TextAreaCounter',
  mixins: [logMixin],
  // 其他所有方法
};
```

如你所见，这个代码片段还添加了一个 `name` 属性，方便辨认调用者。

如果你运行这个包含 mixin 的代码例子，可以看到其日志输出（如图 2-12 所示）。

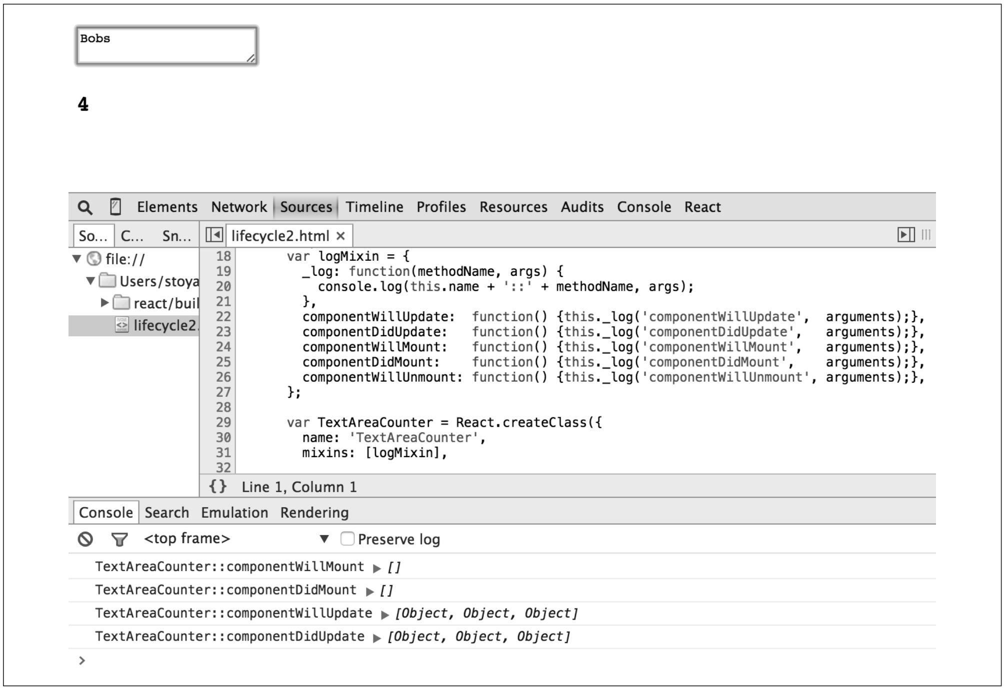



图 2-12：使用 mixin 并辨认组件

# 2.14　生命周期示例：使用子组件

现在，你已经知道如何根据需求去混合和嵌套 React 组件。但目前为止，你只看到了如何在 `render()` 方法中使用 `React.DOM` 组件（而非自定义组件）。接下来，我们看看如何使用一个简单的自定义组件作为子组件。

首先，你可以把计数器部分从组件中分离出来，作为一个单独的新组件：

```
var Counter = React.createClass({
  name: 'Counter',
  mixins: [logMixin],
  propTypes: {
    count: React.PropTypes.number.isRequired,
  },
  render: function() {
    return React.DOM.span(null, this.props.count);
  }
});
```

这个组件仅仅包含了计数器的部分：它渲染一个 `<span>` 标签，没有维护任何 state，仅负责显示父组件提供的 `count` 属性。此外，该组件混合使用了 `logMixin`，会在生命周期方法被调用时输出日志记录。

现在修改父组件 TextAreaCounter 中的 render() 方法。该组件应该选择性地渲染 Counter 组件；如果计数为 0，则不显示数字：

```
render: function() {
  var counter = null;
  if (this.state.text.length > 0) {
    counter = React.DOM.h3(null,
      React.createElement(Counter, {
        count: this.state.text.length,
      })
    );
  }
  return React.DOM.div(null,
    React.DOM.textarea({
      value: this.state.text,
      onChange: this._textChange,
    }),
    counter
  );
}
```

当文本区域为空时，counter 变量的值为 null。当文本框里包含文本时，counter 变量则包含了负责显示字符数量的那部分界面。没有必要在主组件 React.DOM.div 中内联包含整个界面的所有参数。你可以把界面细分为各种小变量，并选择性地使用这些变量。

现在你可以观察到，两个组件的生命周期方法都会被记录下来。图 2-13 显示了从页面加载到改变文本框内容时发生了什么。

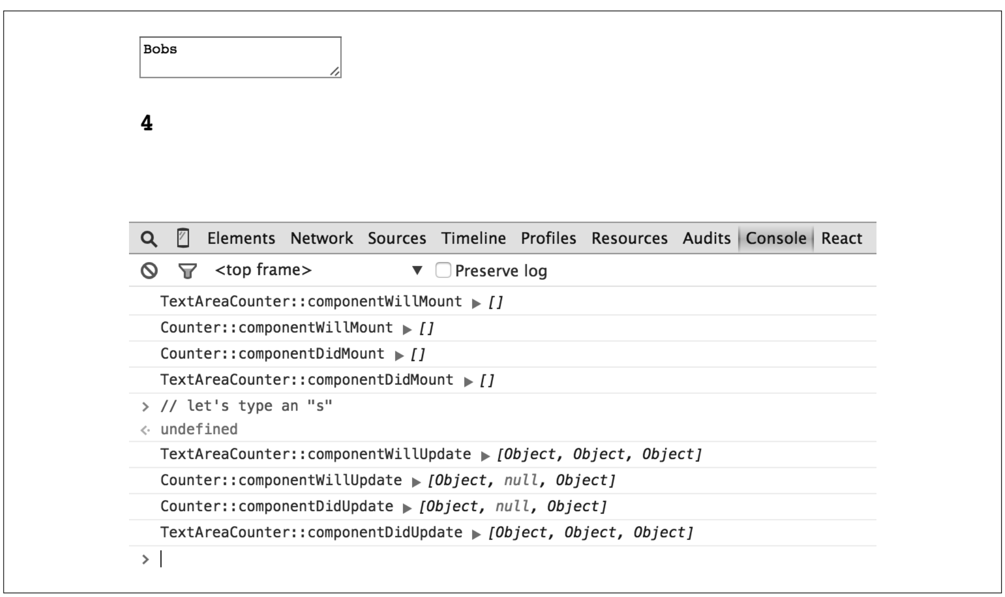



图 2-13：装载并更新两个组件

现在你知道了子组件是如何在父组件之前装载并更新的。

图 2-14 显示了在删除文本框区域的内容后，计数值变为 0 时的情况。在这种情况下，Counter 子节点会变为 null，然后通过 componentWillUnmount 回调函数通知你，随后从 DOM 树中移除原有节点。

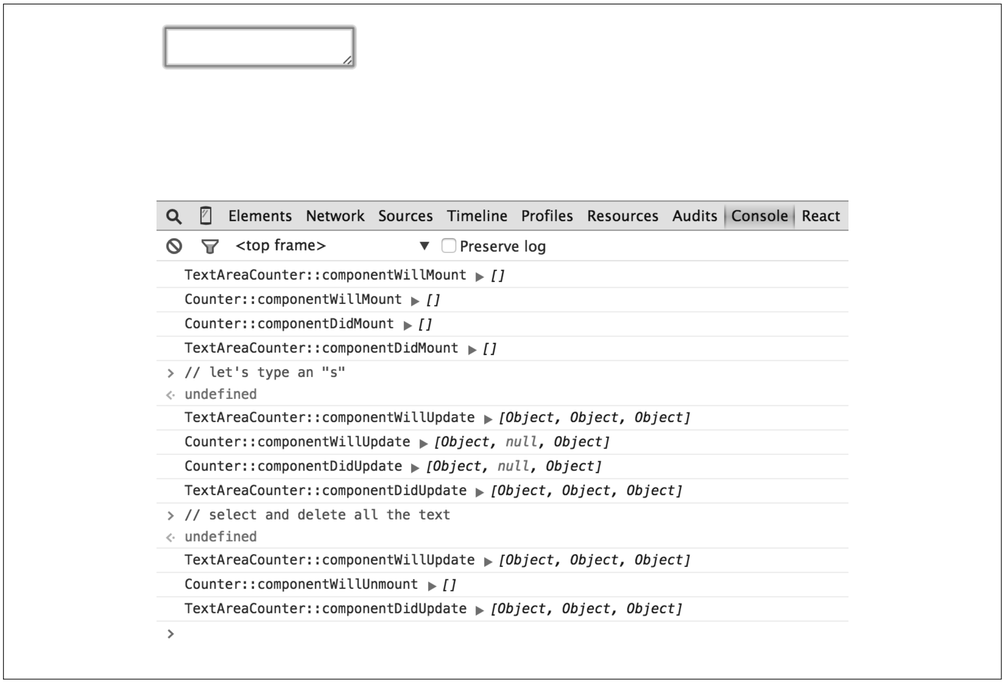



图 2-14：移除计数器组件

## 2.15　性能优化：避免组件更新

最后应该了解的生命周期方法是 shouldComponentUpdate(nextProps, nextState)，它对于构建性能关键型的应用场景尤为重要。这个方法在 componentWillUpdate() 之前调用，给你一个机会进行判断，当非必需时可以取消组件更新。

有一类组件在 render() 方法中只使用了 this.props 和 this.state，而没有其他额外的函数调用。这一类组件被称为“纯”组件。这种组件可以通过实现 shouldComponentUpdate() 方法，用于比较前后的状态与属性。若没有发生任何变化，则返回 false 以节省部分处理能力。此外，对于没有使用 props 和 state 的纯静态组件，直接返回 false 即可。

接下来让我们对 render() 方法的调用情况进行探索，并通过实现 shouldComponentUpdate() 方法优化性能。

首先，打开我们最新的 `Counter` 组件。移除之前用到的日志 mixin，并在调用 `render()` 方法时打印日志到控制台中：

```
var Counter = React.createClass({
  name: 'Counter',
  // mixins: [logMixin],
  propTypes: {
    count: React.PropTypes.number.isRequired,
  },
  render() {
    console.log(this.name + '::render()');
    return React.DOM.span(null, this.props.count);
  }
});
```

对 `TextAreaCounter` 组件进行同样的操作：

```
var TextAreaCounter = React.createClass({
  name: 'TextAreaCounter',
  // mixins: [logMixin],

  // 其他所有方法

  render: function() {
    console.log(this.name + '::render()');
    // 渲染的剩余部分
  }
});
```

现在打开网页，并把字符串 LOL 粘贴到输入框，以替换原有的 Bob，其结果如图 2-15 所示。

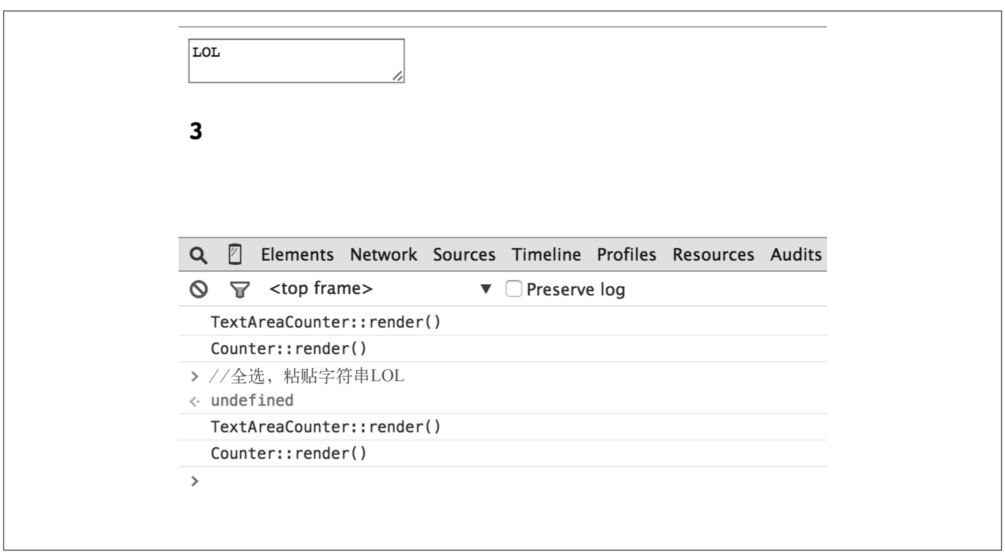



图 2-15：两个组件都被重新渲染

你会发现，改变文本会导致 TextAreaCounter 组件的 render() 方法被调用，随后导致 Counter 组件的 render() 方法被调用。当把 Bob 替换成 LOL 时，字符长度在更新前后不会发生变化，因此计数器的界面不需要更新，没有必要调用 Counter 组件的 render() 方法。在这种场景下，可以通过实现 shouldComponentUpdate() 方法帮助 React 进行优化，在无需额外渲染时，让 shouldComponentUpdate() 返回 false。这个方法接收的参数包括 props 和 state 的新值（在这个组件中无需用到 state），在方法中你可以把当前属性和新的属性进行比较：

```
shouldComponentUpdate(nextProps, nextState_ignore) {
  return nextProps.count !== this.props.count;
},
```

现在再进行同样的操作、把 Bob 替换为 LOL 时，Counter 组件不会被重新渲染了（如图 2-16 所示）。

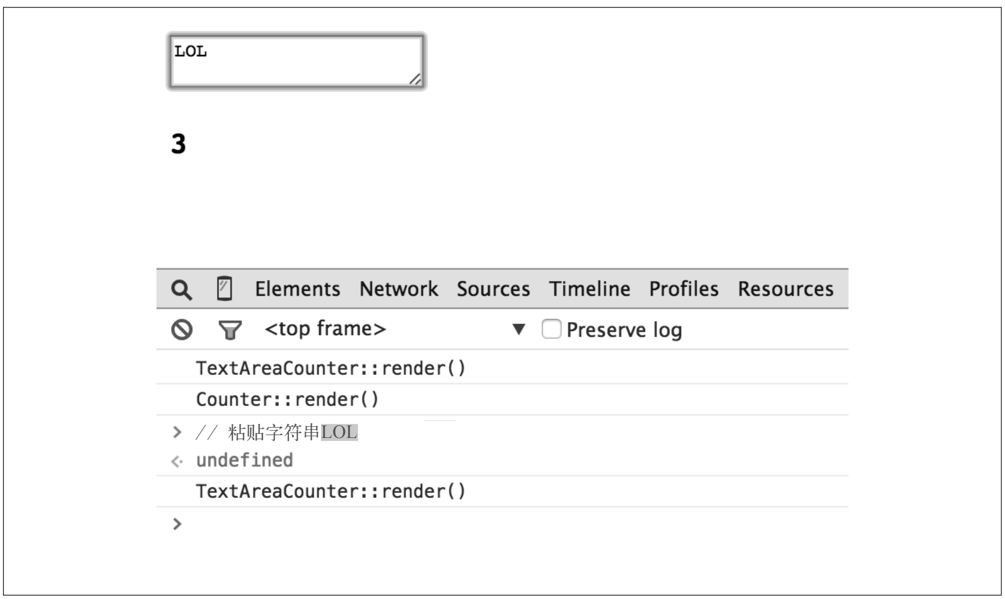



图 2-16：性能优化：少进行一个重新渲染周期

## 2.16 PureRenderMixin

上述 shouldComponentUpdate() 方法的实现相当简单。想要将其变得更为通用也并非难事，你只需要对 this.props 和 nextProps，以及 this.state 和 nextState 进行比较即可。对此，React 通过 mixin 的形式提供了一种通用实现，并且可以简单地应用于任何组件当中。

方法如下：

```
<script src="react/build/react-with-addons.js"></script>
<script src="react/build/react-dom.js"></script>
<script>

  var Counter = React.createClass({
    name: 'Counter',
    mixins: [React.addons.PureRenderMixin],
    propTypes: {
      count: React.PropTypes.number.isRequired,
    },
    render: function() {
      console.log(this.name + '::render()');
      return React.DOM.span(null, this.props.count);
    }
  });

  // ....
</script>
```

结果（如图 2-17 所示）和之前一样：当字符的数量没有发生变化时，`Counter` 组件的 `render()` 方法没有被调用。

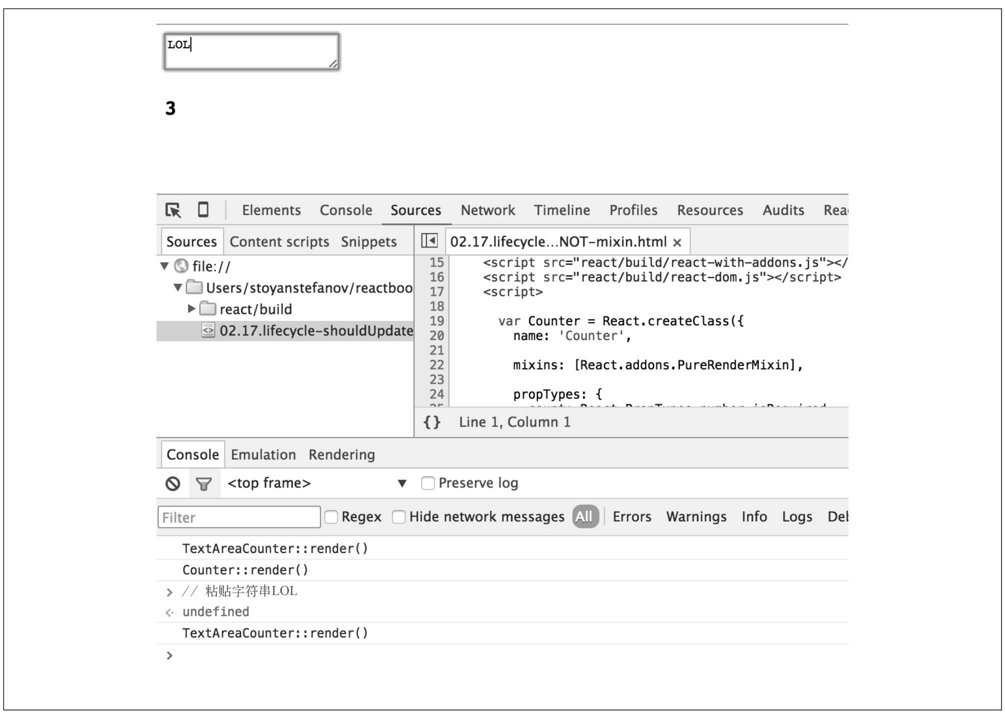



图 2-17：使用 `PureRenderMixin` 轻松完成性能优化

要注意的是，`PureRenderMixin` 并非 React 核心模块的一部分，只是作为一个 React 的插件版本提供。因此为了获取这个 mixin，你需要使用 `react/build/react-with-addons.js`

代替 react/build/react.js。前者提供了一个新的命名空间 React.addons，里面包含了 PureRenderMixin 和其他一些方便使用的插件。[1]

如果你不希望引入全部插件，或者想要实现属于自己的 mixin 版本，不妨参考一下其实现。这个实现其实非常简单直接，仅仅是对相等性进行浅层（非递归）检查：

```
var ReactComponentWithPureRenderMixin = {
  shouldComponentUpdate: function(nextProps, nextState) {
    return !shallowEqual(this.props, nextProps) ||
           !shallowEqual(this.state, nextState);
  }
};
```

注 1：React 的最新版本提供了 React.PureComponent 基础类，因此无需引入该插件。——译者注

第 3 章

# Excel：一个出色的表格组件

到目前为止，你已经学会了如何创建 React 自定义组件，使用普通的 DOM 组件和自定义组件编写（渲染）界面，设置属性，维护状态，挂载组件的生命周期方法，以及通过避免不必要的重新渲染优化性能。

在本章里，我们将温故而知新，创建一个更有趣的组件——数据表格。这个组件有点像 Microsoft Excel v.0.1.beta 版本的原型，你可以对数据表的内容进行编辑、排序、搜索（筛选），并以可下载的文件格式导出数据。

## 3.1　构造数据

表格和数据是紧密联系的，因此这个表格组件（不如把它称作 Excel 吧）需要接收一个数据数组和一个表头数组。为了方便测试，我们从维基百科（http://en.wikipedia.org/wiki/List_of_bestselling_books）抓取了一份畅销书列表：

```
var headers = [
  "Book", "Author", "Language", "Published", "Sales"
];

var data = [
  ["The Lord of the Rings", "J. R. R. Tolkien",
    "English", "1954-1955", "150 million"],
  ["Le Petit Prince (The Little Prince)", "Antoine de Saint-Exupéry",
    "French", "1943", "140 million"],
  ["Harry Potter and the Philosopher's Stone", "J. K. Rowling",
    "English", "1997", "107 million"],
  ["And Then There Were None", "Agatha Christie",
```

```
    "English", "1939", "100 million"],
  ["Dream of the Red Chamber", "Cao Xueqin",
    "Chinese", "1754-1791", "100 million"],
  ["The Hobbit", "J. R. R. Tolkien",
    "English", "1937", "100 million"],
  ["She: A History of Adventure", "H. Rider Haggard",
    "English", "1887", "100 million"],
];
```

## 3.2 表头循环

第一步，我们只需要把表格头部显示出来。以下是一个大致骨架：

```
var Excel = React.createClass({
  render: function() {
    return (
      React.DOM.table(null,
        React.DOM.thead(null,
          React.DOM.tr(null,
            this.props.headers.map(function(title) {
              return React.DOM.th(null, title);
            })
          )
        )
      )
    );
  }
});
```

现在你已经有了一个可用的组件，具体用法如下：

```
ReactDOM.render(
  React.createElement(Excel, {
    headers: headers,
    initialData: data,
  }),
  document.getElementById("app")
);
```

输出结果如图 3-1 所示。




图 3-1：渲染表头

值得注意的是，这里用到了数组类型的 map() 方法，其用途是返回一个包含子节点组件的新数组。数组类型的 map() 方法会把数组（在这个例子中是 header 数组）中的每个元素都传递到回调函数中。在这里，回调函数创建一个新的 <th> 组件并将其作为函数返回值。

这就是 React 之美：你可以使用 JavaScript 语法，并借助这门语言的全部特性去创建界面。循环、条件判断等功能都可以在 React 中使用，你无需学习另一门“模板”语言或语法即可创建界面。

在传递子节点给组件时，既可以传递一个单独的数组参数，也可以把每个子节点作为独立的参数传递。因此下列两种写法是等价的：

```
// 传递独立的参数
React.DOM.ul(
  null,
  React.DOM.li(null, 'one'),
  React.DOM.li(null, 'two')
);

// 传递数组
```

```
React.DOM.ul(
  null,
  [
    React.DOM.li(null, 'one'),
    React.DOM.li(null, 'two')
  ]
);
```

## 3.3 消除控制台的警告信息

图 3-1 中的控制台截图显示了一个警告信息。这个信息是关于什么的？应该如何修复呢？它的意思是："数组或迭代器中的每个子节点应包含唯一的 key 属性。请检查使用了 <tr> 的顶层渲染调用。"

使用了 <tr> 的顶层渲染调用？由于这个应用现在只有一个组件，不难推断出问题出现的地方，但在现实开发中，可能有许多组件都创建了 <tr> 元素。Excel 仅仅是一个变量名，在 React 外部被赋值给一个 React 组件而已，因此 React 并不知道这个组件的名字叫 Excel。你可以通过给组件声明一个 displayName 属性来解决这个问题：

```
var Excel = React.createClass({
  displayName: 'Excel',
  render: function() {
    // ...
  }
};
```

现在 React 可以识别问题出在哪里了，警告信息变为"Each child in an array should have a unique"key" prop. Check the render method of `Excel`."。这样调试就方便多了，但警告信息依然存在。为了修复这个问题，只需要根据警告去检查代码即可。现在你已经知道问题发生在哪个 render() 函数中：

```
this.props.headers.map(function(title, idx) {
  return React.DOM.th({key: idx}, title);
})
```

这个函数做了什么事情？传递给 Array.prototype.map() 的回调函数被调用时会提供三个参数：元素的值、元素的索引值（0、1、2 等）和整个数组。你只需要把每个元素的索引值（idx）提供给 React 作为 key 属性即可。这个 key 属性只需要在该数组中保持唯一，而不需要保证在整个 React 应用中唯一。

现在 key 的问题已经被修复了，加上一点 CSS 进行美化，你就可以享受这个新组件的 0.0.1 版本了。它看起来挺漂亮的，而且控制台警告已经消失了（如图 3-2 所示）。

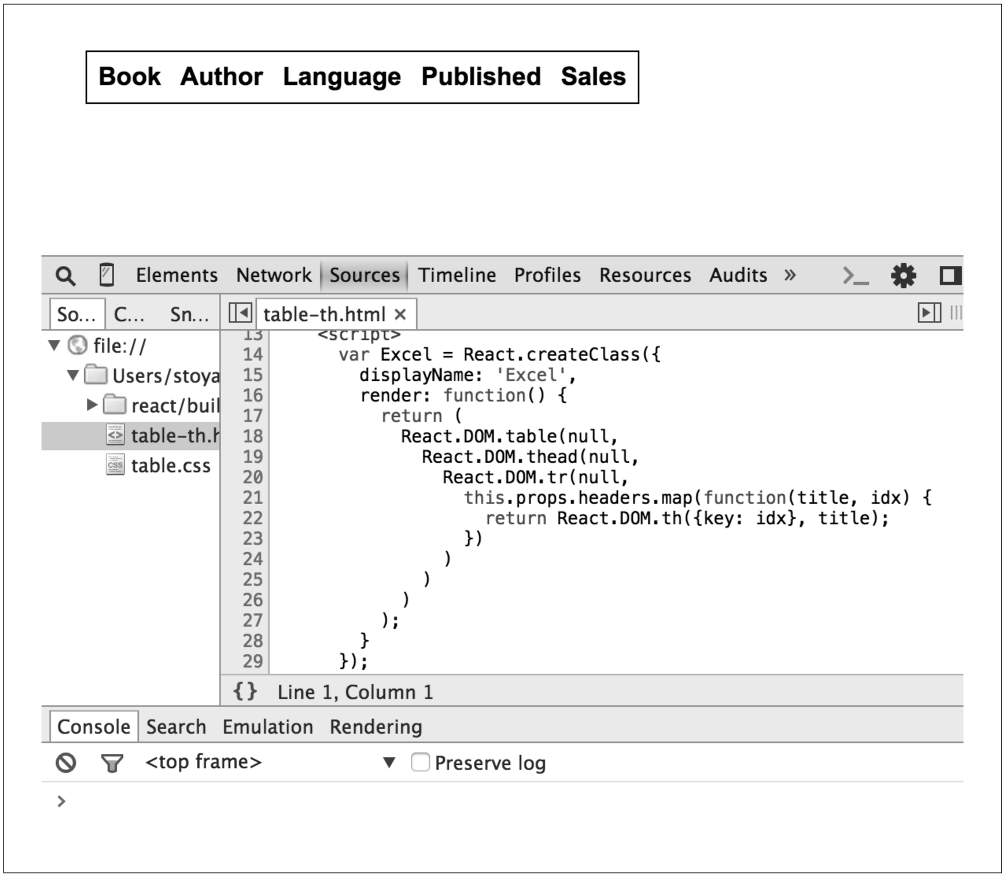



图 3-2：渲染没有警告的表格组件

> 仅仅为了调试而添加 displayName 的做法似乎有些麻烦，不过还有更方便的方法：使用 JSX（会在第 4 章讨论）就不需要再定义这个属性，名字会自动产生。

## 3.4　添加 <td> 内容

既然已经完成了漂亮的表头部分，现在是时候添加表格内容了。表头中的内容是一个一维数组（单行），但是 data 是二维的。因此你需要双重循环：外层循环遍历行，而内层要经过每一行中的每一个数据（单元格）。这可以通过前面提到的 .map() 方法完成：

```
data.map(function(row) {
  return (
    React.DOM.tr(null,
      row.map(function(cell) {
        return React.DOM.td(null, cell);
      })
    )
  );
})
```

此外，还需要考虑 data 变量本身：数据从哪里来？如何更改数据？ Excel 组件调用者应该可以传递初始的表格数据。但随后数据可能会发生变化，因为用户可能会对表格进行排序、编辑等操作。换句话说，组件的 state 会发生变化。因此，我们使用 this.state.data 保持跟踪数据变化，使用 this.props.initialData 进行组件初始化。目前完整的实现看起来类似于下面这样（结果如图 3-3 所示）：

```
getInitialState: function() {
  return {data: this.props.initialData};
},

render: function() {
  return (
    React.DOM.table(null,
      React.DOM.thead(null,
        React.DOM.tr(null,
          this.props.headers.map(function(title, idx) {
            return React.DOM.th({key: idx}, title);
          })
        )
      ),
      React.DOM.tbody(null,
        this.state.data.map(function(row, idx) {
          return (
            React.DOM.tr({key: idx},
              row.map(function(cell, idx) {
                return React.DOM.td({key: idx}, cell);
              })
            )
          );
        })
      )
    )
  );
}
```

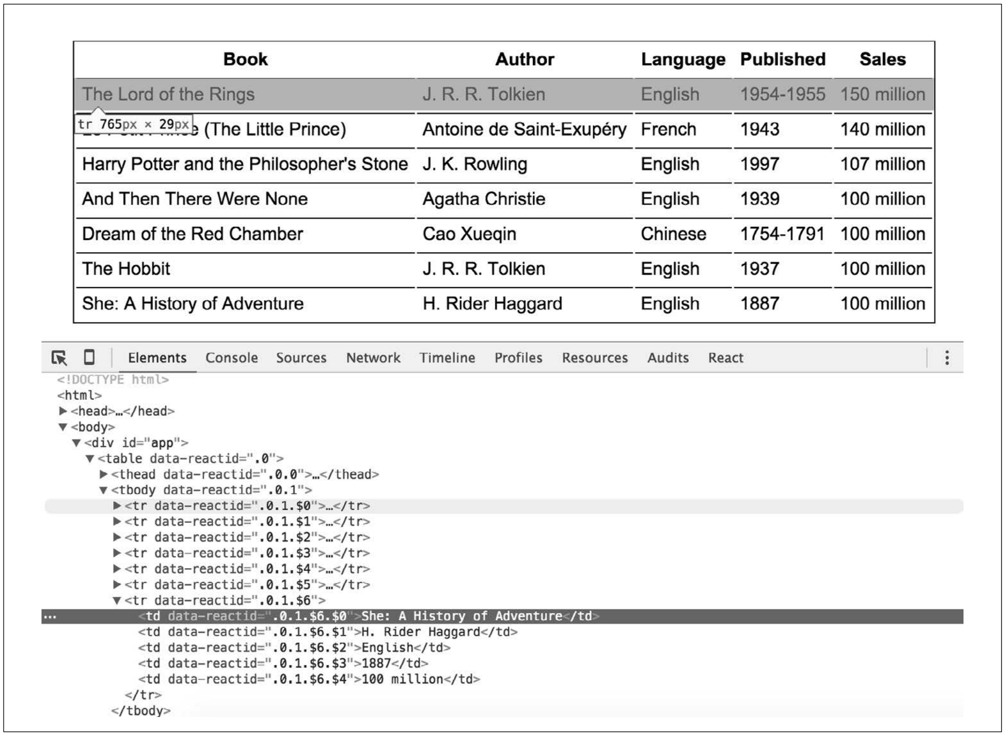



图 3-3：渲染整个表格

可以看到，重复 `{key: idx}` 使得组件数组中的每个元素都拥有唯一的键名。虽然所有的 `.map()` 循环都是从索引 0 开始的，但是没有关系，因为 `key` 只需要保证在当前循环中唯一，而不是在整个应用中唯一。

> `render()` 函数现在已经有点复杂、难以理解了，特别是闭合括号 `}` 和 `)`。别担心，JSX 可以帮助你减轻痛苦！

前面的代码片段中省略了 `propTypes` 属性（尽管这个属性是可选的，但是推荐使用）。该属性既可以用作数据验证，也可以为组件提供文档。具体到这个例子，我们要尽量减少用户提供垃圾数据到这个漂亮 Excel 组件的可能性。`React.PropTypes` 提供了一个 `array` 验证器，以确保属性总是一个数组。此外，它还提供了一个 `arrayOf` 函数，以验证数组元素的具体类型。在这个例子中，我们让表头标题和表格数据都只接受字符串数组：

```
propTypes: {
  headers: React.PropTypes.arrayOf(
    React.PropTypes.string
```

```
    ),
    initialData: React.PropTypes.arrayOf(
      React.PropTypes.arrayOf(
        React.PropTypes.string
      )
    ),
  },
```

现在数据检查足够严格了！

## 如何改进该组件

对于一个普通的 Excel 电子表格而言，只能接受字符串数据显得过于苛刻。作为练习，你可以允许其接受更多数据类型（`React.PropTypes.any`）并根据不同类型进行不同方式的渲染（例如，右对齐数字类型的数据）。

# 3.5　排序

当你看见网页中的表格时，是否想对其进行各种各样的排序呢？幸运的是，这个需求可以用 React 轻松完成。事实上，这个例子恰恰体现了 React 的闪光之处，因为你要做的仅仅是对数据数组进行排序，而 React 将帮你完成界面的更新。

首先，在表头处添加一个点击事件处理器：

```
React.DOM.table(null,
  React.DOM.thead({onClick: this._sort},
    React.DOM.tr(null,
    // ...
```

现在，让我们实现 `_sort` 函数。首先你要知道根据哪一列进行排序。这可以通过触发事件的对象（在这里是表头 `<th>` 标签）的 `cellIndex` 属性取得：

```
var column = e.target.cellIndex;
```

> 你可能很少见到有人在应用开发中使用 `cellIndex` 属性。这是一个早在 DOM Level 1 就被定义的属性，其定义为 " 这个单元格在该行中的索引值 "。它随后在 DOM Level 2 中被定义为只读属性。

你还需要一份要排序的数据副本。如果你直接使用数据的 `sort()` 方法，会直接修改原数组，也就意味着 `this.state.data.sort()` 会修改 `this.state`。你已经知道 `this.state` 是不应该被直接修改的，只能通过 `setState()` 修改：

```
// 复制数据
var data = this.state.data.slice(); // 或者使用ES6中的Array.from(this.state.data)
```

接下来，排序功能可以通过 `sort()` 方法中传入的回调函数完成：

```
data.sort(function(a, b) {
  return a[column] > b[column] ? 1 : -1;
});
```

最后，通过 `setState()` 设置排序后的新数据：

```
this.setState({
  data: data,
});
```

现在，当你点击表头时，内容会按照字母顺序排序（如图 3-4 所示）。

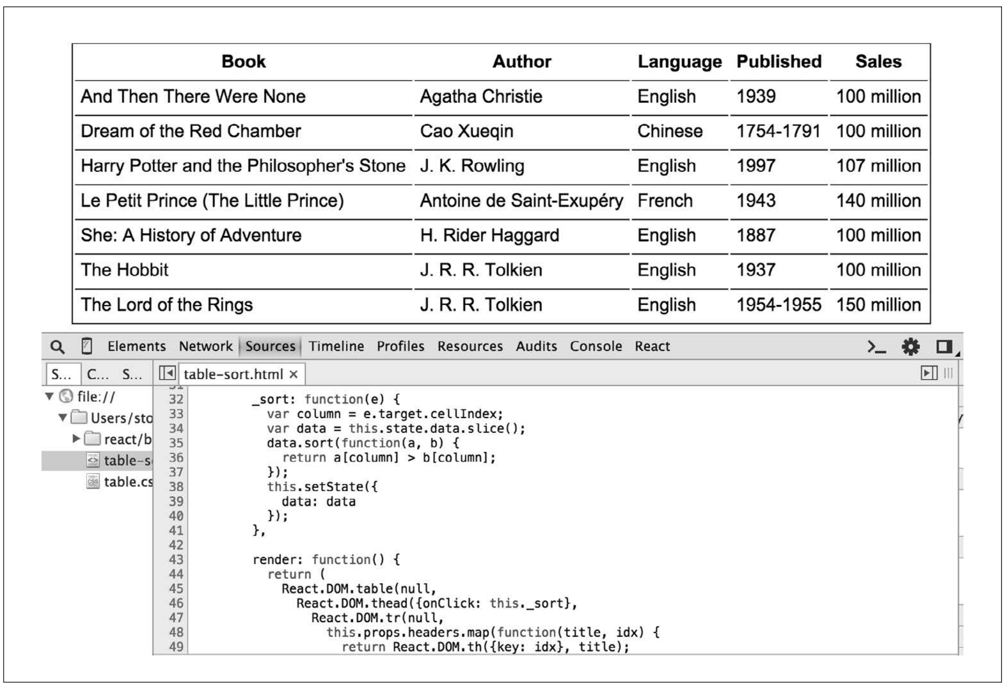



图 3-4：按照书名排序

就是这么简单，你完全不需要关心视图渲染。在 `render()` 方法中，你已经一次性为组件定义了在给定某些数据时如何进行展现。当数据发生改变时，视图也会对应更新；但你不再需要关心它是如何变化的了。

## 如何改进该组件

这里用到的排序逻辑相当简单，仅仅满足了关于 React 话题的讨论。你可以让它变得更加出色：比如对内容进行解析；当其值是数字时，可以选择带或不带单位等。

# 3.6 排序的视觉提示

现在表格已经排好序了，但用户不能清晰地看到表格是根据哪一列进行排序的。让我们更新一下界面，在进行排序的那一列显示一个箭头符号。与此同时，我们还将实现降序排列的功能。

为了跟踪新的状态，你需要添加以下两个新属性。

- `this.state.sortby`

  当前被选择的列索引值
- `this.state.descending`

  一个布尔类型的标记，决定按照升序还是降序排列

```
getInitialState: function() {
  return {
    data: this.props.initialData,
    sortby: null,
    descending: false,
  };
},
```

在 `_sort()` 函数中，你可以指定采用其中哪种方法进行排序。默认为升序排列；在当前选中的列索引和之前一样并且当前还不是降序排列时，则进行降序排列：

```
var descending = this.state.sortby === column && !this.state.descending;
```

你还需要对排序的回调函数进行一点调整：

```
data.sort(function(a, b) {
  return descending
    ? (a[column] < b[column] ? 1 : -1)
    : (a[column] > b[column] ? 1 : -1);
});
```

最后，你需要设置新的 state：

```
this.setState({
  data: data,
  sortby: column,
  descending: descending,
});
```

还剩一件事情需要完成，那就是修改 `render()` 函数，指出当前的排序方向。对于当前已排

好序的那一列，让我们在标题上添加一个箭头符号：

```
this.props.headers.map(function(title, idx) {
  if (this.state.sortby === idx) {
    title += this.state.descending ? ' \u2191' : ' \u2193'
  }
  return React.DOM.th({key: idx}, title);
}, this)
```

现在我们完成了排序功能。用户可以对任意列进行排序：首次点击时对该列进行升序排列，再点击一次可以进行降序排列；而视图会给出对应的视觉提示（如图 3-5 所示）。

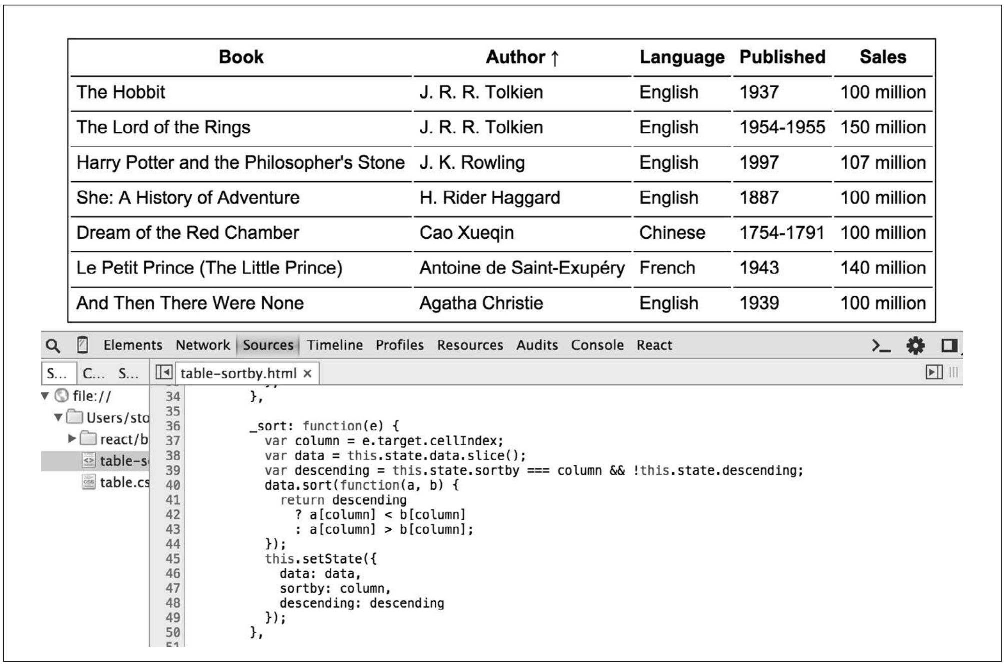



图 3-5：升序 / 降序排列

# 3.7　编辑数据

接下来，我们要让用户可以在这个 Excel 表格组件中编辑数据。下面介绍的解决方案包括三个步骤。

(1) 双击一个单元格。Excel 找出点击的单元格，并把表格内容从简单的文本变为一个预先填充了原内容的输入框（如图 3-6 所示）。

| Book | Author | Language | Published | Sales |
|---|---|---|---|---|
| The Lord of the Rings | J. R. R. Tolkien | English | 1954-1955 | 150 million |

图 3-6：双击单元格时，内容变为一个数据框

(2) 编辑内容（如图 3-7 所示）。

| Book | Author | Language | Published | Sales |
|---|---|---|---|---|
| The Lord of the Rings | J. R. R. Tolkien | English | 1954-1955 | 200 million |
| Le Petit Prince (The Little Prince) | Antoine de Saint-Exupéry | French | 1943 | 140 million |

图 3-7：编辑内容

(3) 敲下回车。输入框消失，表格数据更新为新输入内容（如图 3-8 所示）。

| Book | Author | Language | Published | Sales |
|---|---|---|---|---|
| The Lord of the Rings | J. R. R. Tolkien | English | 1954-1955 | 200 million |
| Le Petit Prince (The Little Prince) | Antoine de Saint-Exupéry | French | 1943 | 140 million |

图 3-8：敲下回车后，修改原内容

## 3.7.1 可编辑单元格

首先要设置一个简单的事件监听器。当用户双击时，组件“记住”用户选择的单元格：

```
React.DOM.tbody({onDoubleClick: this._showEditor}, ....)
```

这里并没有使用 W3C 规范的 `ondblclick`，而是使用了更友好、更易于阅读的 `onDoubleClick`。

`_showEditor` 函数就像下面这样：

```
_showEditor: function(e) {
  this.setState({edit: {
    row: parseInt(e.target.dataset.row, 10),
    cell: e.target.cellIndex,
  }});
},
```

这里发生了什么？

- 这个函数设置了 this.state 中的 edit 属性。该属性在进入编辑状态之前是 null，之后则变为一个包含了属性 row 和 cell 的对象，这两个属性分别代表被编辑单元格的行索引值和列索引值。所以如果你点击的是最开始的单元格，this.state.edit 的值就是 {row: 0, cell: 0}。
- 为了算出单元格的索引值，和之前一样，你需要使用 e.target.cellIndex，其中 e.target 指向的是被双击的单元格 <td>。
- 在 DOM 结构中没有直接获取 rowIndex 的方法，你需要自己设定一个 data- 自定义属性。每个单元格都应该包含 data-row 属性，并设置为行索引值，然后你可以使用 parseInt() 取得该索引值。

最后，还有一些说明和先决条件。首先，edit 属性在一开始是不存在的，所以要在 getInitialState() 方法中进行初始化。现在的 getInitialState() 方法应该如下所示：

```
getInitialState: function() {
  return {
    data: this.props.initialData,
    sortby: null,
    descending: false,
    edit: null, // {row: index, cell: index}
  };
},
```

data-row 属性用于跟踪行索引。现在 tbody() 的整个结构就像这样：

```
React.DOM.tbody({onDoubleClick: this._showEditor},
  this.state.data.map(function(row, rowidx) {
    return (
      React.DOM.tr({key: rowidx},
        row.map(function(cell, idx) {
          var content = cell;

          // TODO - 如果idx和rowidx的值与当前单元格匹配,
          // 则把content变为一个输入框;否则只需展示文本内容

          return React.DOM.td({
            key: idx,
            'data-row': rowidx
          }, content);
        }, this)
      )
    );
  }, this)
)
```

最后需要完成 TODO 中的未实现功能。我们需要在特定条件下生成一个文本框区域。由于调用 setState() 方法设置了 edit 属性，整个 render() 方法会被再次调用。React 对表格进行重新渲染，给了你一个在双击表格时更新单元格内容的机会。

### 3.7.2 输入字段的单元格

现在把 TODO 注释替换为具体代码。首先，我们需要取得编辑状态的 state 值：

```
var edit = this.state.edit;
```

判断 edit 属性是否已设置。如果是，则判断当前单元格是否为待编辑单元格：

```
if (edit && edit.row === rowidx && edit.cell === idx) {
  // ...
}
```

如果匹配到目标单元格，就创建一个表单和一个输入框，并把内容填充到输入框中：

```
content = React.DOM.form({onSubmit: this._save},
  React.DOM.input({
    type: 'text',
    defaultValue: content,
  })
);
```

如你所见，这个表单仅有一个输入框，输入框里预填充了单元格的文本内容。当提交表单时，会进入私有方法 _save() 的调用中。

### 3.7.3 保存

要完成编辑功能，还需要在用户完成输入并提交表单（通过按下回车键）时，保存内容更改：

```
_save: function(e) {
  e.preventDefault();
  // 进行保存
},
```

在避免浏览器默认行为后（目的是避免网页重新加载），你需要取得一个输入框的引用：

```
var input = e.target.firstChild;
```

复制一份原有数据，避免直接操作 this.state：

```
var data = this.state.data.slice();
```

使用新的值修改原有数据。单元格的行索引和列索引可以通过 state 的 edit 属性取得：

```
data[this.state.edit.row][this.state.edit.cell] = input.value;
```

最后更新 state，使视图重新渲染：

```
this.setState({
  edit: null, // 完成编辑
  data: data,
});
```

## 3.7.4 结论与虚拟 DOM Diff 算法

现在，编辑功能已经完成了。这个功能不需要我们编写太多代码，你需要做的仅仅是：

- 通过 `this.state.edit` 属性跟踪需要编辑的单元格；
- 在渲染时，如果单元格的行列索引匹配到用户双击的单元格，则在该单元格中显示输入框；
- 从输入框获取新输入的值，更新表格数据的数组。

当你使用 `setState()` 方法更新数据时，React 将会调用组件的 `render()` 方法，让界面如魔术般地更新。也许仅仅因为一个单元格的变化而更新整个表格看起来不够高效，但 React 实际上只会修改一个单元格。

如果你打开浏览器开发工具，可以看到当你和应用进行交互时 DOM 树的哪个部分得到了更新。在图 3-9 中可以看到，当把 *The Lord of the Rings* 的语言一栏从 English 改为 Engrish 时，开发者工具会高亮显示 DOM 结构的改变。

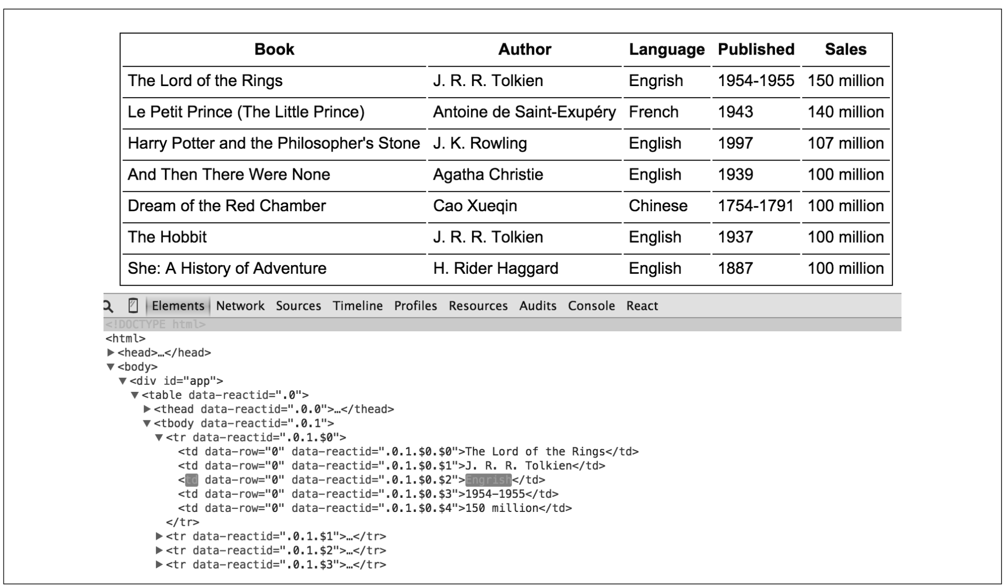



图 3-9：高亮显示 DOM 结构的改变

在幕后，React 调用了 `render()` 方法，并根据预期 DOM 结果创建了一个轻量级的树形表达。这被称为虚拟 DOM 树。当再次调用 `render()` 方法时（比如在调用 `setState()` 方法之

后），React 对于前后的虚拟树作出比较，计算出 diff。基于 diff，React 可以计算出改变浏览器 DOM 结构所需的最小 DOM 操作（比如 `appendChild()`、`textContent` 等）。

在图 3-9 中，单元格只发生了一处改变，因此没有必要重新渲染整个表格。React 通过计算改变的最小集合，批量进行 DOM 操作。众所周知，DOM 操作是很耗时的（相比于纯 JavaScript 操作、函数调用等），并且通常会成为大型 Web 应用程序渲染性能的瓶颈，因此 React 会尽量减少 DOM 操作。

简而言之，对于性能和界面更新，React 通过以下方式给予你支持：

- 轻量级操作 DOM
- 使用事件委托响应用户交互

# 3.8 搜索

下一步，我们为 `Excel` 组件增加一个搜索功能，允许用户过滤表格内容。我们计划这样做：

- 增加一个按钮作为这个新功能的开关（如图 3-10 所示）；

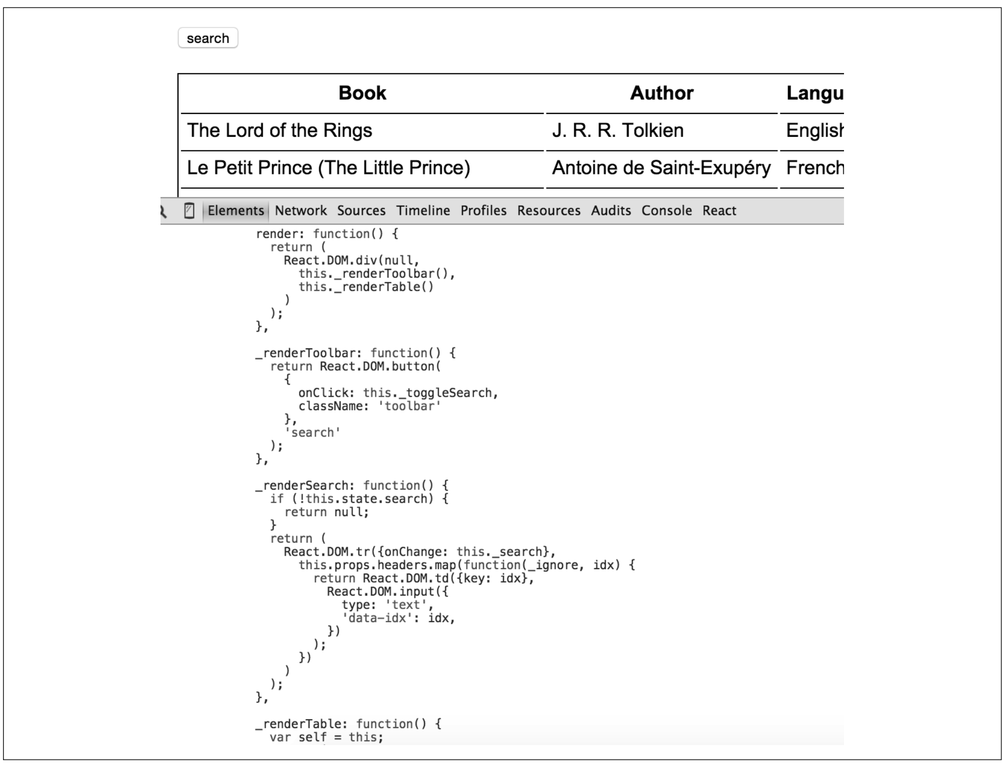



图 3-10：搜索按钮

- 如果打开搜索功能，则在表格中新增一行输入框，每个输入框各自负责搜索对应的列（如图 3-11 所示）；

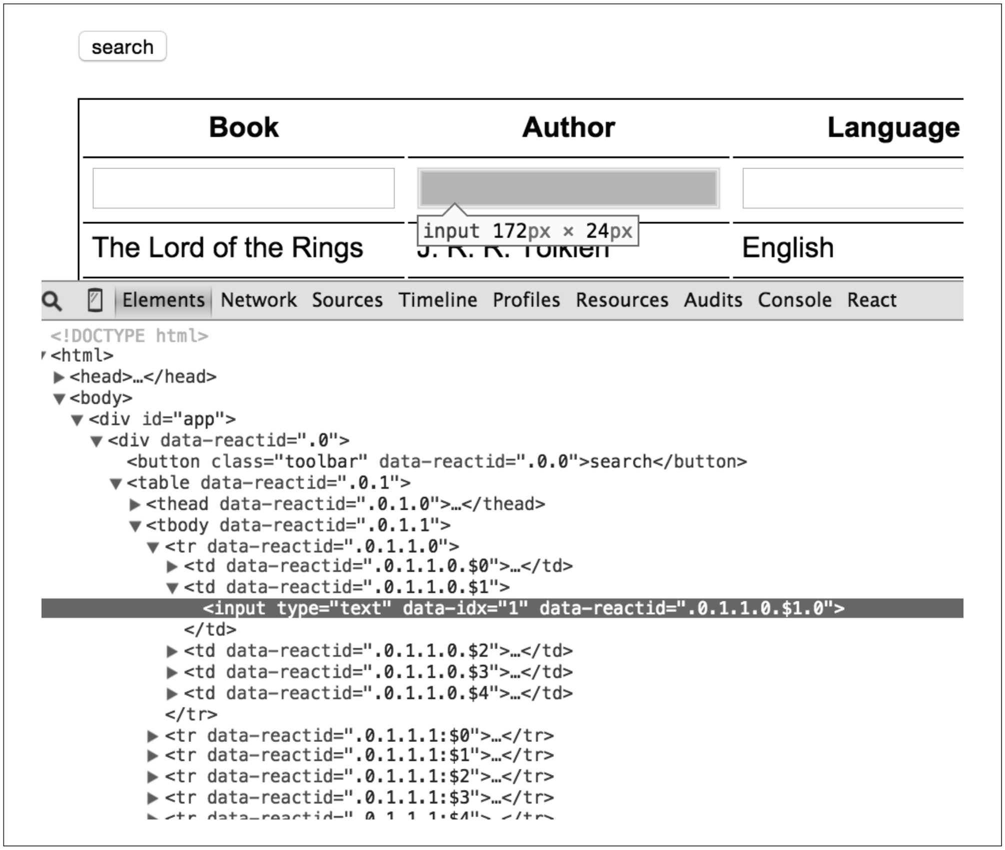
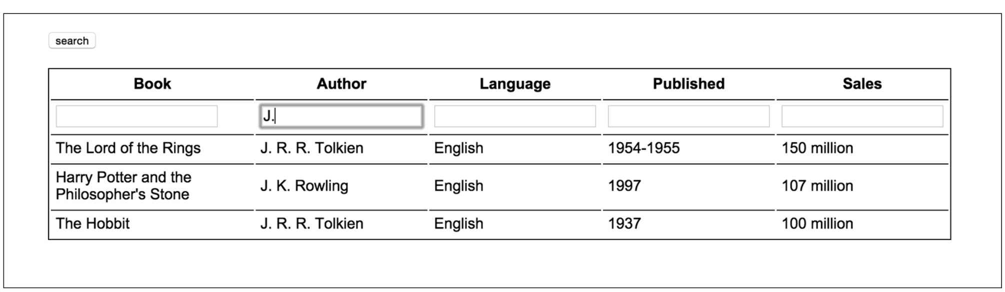



图 3-11：一行用于搜索 / 筛选的输入框

- 当用户在输入框中输入内容时，对 `state.data` 数组进行筛选，只显示匹配的内容（如图 3-12 所示）。




| Book | Author | Language | Published | Sales |
|---|---|---|---|---|
| | J. | | | |
| The Lord of the Rings | J. R. R. Tolkien | English | 1954-1955 | 150 million |
| Harry Potter and the Philosopher's Stone | J. K. Rowling | English | 1997 | 107 million |
| The Hobbit | J. R. R. Tolkien | English | 1937 | 100 million |

图 3-12：搜索结果

### 3.8.1 状态与界面

首先，要给 `this.state` 对象添加一个 `search` 属性，以追踪搜索功能是否打开：

```
getInitialState: function() {
  return {
    data: this.props.initialData,
    sortby: null,
    descending: false,
    edit: null, // [行索引，列索引],
    search: false,
  };
},
```

下一步是更新界面。为了让代码变得更加易于管理，我们要把 `render()` 函数分隔成若干个专用的小代码块。目前 `render()` 函数只渲染了一个表格。我们把它重命名为 `_renderTable()` 方法。接下来，搜索按钮将成为整个工具栏的一部分（很快还要添加一个"输出"功能），因此我们把按钮的渲染放在 `_renderToolbar()` 中，作为该方法的一部分。

目前的代码如下所示：

```
render: function() {
  return (
    React.DOM.div(null,
      this._renderToolbar(),
      this._renderTable()
    )
  );
},

_renderToolbar: function() {
  // TODO
},

_renderTable: function() {
  // 和之前的render()方法功能相同
},
```

如你所见，新的 `render()` 函数返回一个 `div` 组件，包含两个子元素：工具栏和表格。你已经知道表格是如何渲染的了，而且工具栏目前仅需渲染一个按钮：

```
_renderToolbar: function() {
  return React.DOM.button(
    {
      onClick: this._toggleSearch,
      className: 'toolbar',
    },
    'search'
  );
},
```

如果开启搜索功能（意味着 this.state.search 的值被设置为 true），你需要渲染一行输入框组件。我们使用 _renderSearch() 来处理这件事情：

```
_renderSearch: function() {
  if (!this.state.search) {
    return null;
  }
  return (
    React.DOM.tr({onChange: this._search},
      this.props.headers.map(function(_ignore, idx) {
        return React.DOM.td({key: idx},
          React.DOM.input({
            type: 'text',
            'data-idx': idx,
          })
        );
      })
    )
  );
},
```

如你所见，搜索功能关闭时，这个函数不需要渲染任何内容，因此函数返回 null。当然还有另一种方法，就是让函数调用者根据搜索开关决定是否调用该函数。不过相比之下，前者有助于简化 _renderTable() 函数。现在只需要对 _renderTable() 函数进行如下修改即可。



修改前：

```
React.DOM.tbody({onDoubleClick: this._showEditor},
  this.state.data.map(function(row, rowidx) { // ...
```

修改后：

```
React.DOM.tbody({onDoubleClick: this._showEditor},
  this._renderSearch(),
  this.state.data.map(function(row, rowidx) { // ...
```

搜索输入框仅仅是 data 主循环（负责创建表格的所有行列）前的另一个子节点。当 _renderSearch() 方法返回 null 时，React 简单地跳过这个额外子节点，并进行接下来的表格渲染。

目前我们已经完成了搜索功能的界面修改。接下来要关注搜索功能的实际业务逻辑了。

## 3.8.2　筛选内容

我们打算把搜索功能做得非常简单：接收一个内容数组，对其调用 Array.prototype.filter() 方法进行筛选，然后返回一个经过过滤的数组，包含符合搜索字符串的所有元素。

界面依然使用 this.state.data 进行渲染，但 this.state.data 的内容会被过滤掉一部分。

你需要在搜索之前复制一份完整的数据，避免在搜索之后丢失数据。这样用户可以回到整个表格中，也可以改变搜索关键词以获取不同的匹配数据。我们把这份数据的副本（实际上是一份引用）称为 _preSearchData：

```
var Excel = React.createClass({
  // 业务逻辑

  _preSearchData: null,

  // 更多逻辑
});
```

当用户点击 search 按钮进行搜索时，_toggleSearch() 方法会被调用。这个函数负责切换搜索功能的打开与关闭。具体逻辑如下：

- 根据开关设置 this.state.search 的值为 true 或者 false；
- 在开启搜索功能时，“记住”原有数据；
- 在关闭搜索功能时，恢复原有数据。

这个函数的具体逻辑如下所示：

```
_toggleSearch: function() {
  if (this.state.search) {
    this.setState({
      data: this._preSearchData,
      search: false,
    });
    this._preSearchData = null;
  } else {
    this._preSearchData = this.state.data;
    this.setState({
      search: true,
    });
  }
},
```

最后还需要实现 _search() 函数。每当搜索行中的内容发生改变，也就是用户在其中一个输入框里输入内容时，该函数会被调用。以下是函数的完整实现，以及一些详细说明：

```
_search: function(e) {
  var needle = e.target.value.toLowerCase();
  if (!needle) { // 当搜索字符串被删除时
    this.setState({data: this._preSearchData});
    return;
  }
  var idx = e.target.dataset.idx; // 需要搜索的那一列的索引值
  var searchdata = this._preSearchData.filter(function(row) {
    return row[idx].toString().toLowerCase().indexOf(needle) > -1;
```

```
  });
  this.setState({data: searchdata});
},
```

你可以通过变化的事件目标（输入框）获取用户输入的字符串：

```
var needle = e.target.value.toLowerCase();
```

如果搜索字符串为空（用户删除了输入的内容），函数会把 state 修改为缓存中的原始数据：

```
if (!needle) {
  this.setState({data: this._preSearchData});
  return;
}
```

如果搜索字符串非空，则过滤原始数据，并把过滤结果作为 `data` 的新 state：

```
var idx = e.target.dataset.idx;
var searchdata = this._preSearchData.filter(function(row) {
  return row[idx].toString().toLowerCase().indexOf(needle) > -1;
});
this.setState({data: searchdata});
```

至此，搜索功能已经完成了。要实现该功能，你只需要：

- 增加搜索界面；
- 根据需要显示 / 隐藏新界面内容；
- 实际“业务逻辑”，即一个简单的数组 `filter()` 方法调用。

原本的表格渲染逻辑并不需要改变。和往常一样，你只需要关心数据的状态；不管数据状态如何改变，都只需要把渲染过程（以及所有繁杂的 DOM 操作）交给 React 实现就可以了。

### 3.8.3 如何改进搜索功能

这仅仅是一个用作演示的简单例子。不妨思考一下：如何改进搜索功能？

切换搜索按钮的标签文字是一项可改进的工作。比如当搜索功能打开时（`this.state.search === true`），提示用户“搜索完成”。

另一项可以尝试的工作是使用多个搜索框实现*多重搜索*，也就是筛选已经筛选过的数据。如果用户在语言一栏键入 Eng，然后在另一个搜索框中输入内容，为何不能只在之前的搜索结果之上继续搜索呢？你会如何实现这个功能？

# 3.9 即时回放

如你所见，组件会关心自身的 state，并让 React 选择适当的时机进行渲染和重新渲染。这意味着，在给定相同数据（state 和 props）的情况下，无论这个特定数据状态的前后发生了什么改变，应用看起来都应该是完全一样的。这个特性使得你可以重现一些外界出现的问题并进行调试。

假设用户在使用你的应用时遇到问题，他们只需要点击一个按钮就能完成问题上报，而无需解释当时发生的事情。这份问题报告只需要把 `this.state` 和 `this.props` 发回给你，你就可以重现实际对应的应用状态，并看到视觉效果。

React 会通过同一个 state 和 props 渲染出同一份应用界面。基于这个事实，我们还可以开发出“撤销”功能。实际上，这个功能的实现非常简单：只需要还原之前的 state 就可以了。

有了这个想法，让我们更进一步，做点更好玩的事情。每当 `Excel` 组件的 state 发生改变时，我们都把 state 记录下来，并进行回放。让 state 在你面前一一回放出来将会非常有趣。

从实现上看，我们不关注改变是何时发生的，只需要每隔一秒种“回放”一次应用的更改。要实现该功能，我们添加一个 `_logSetState()` 方法，搜索原有的 `setState()` 调用并全部替换为这个新方法。

因此，所有的 `this.setState(newSate);` 都会变为 `this._logSetState(newState);`。`_logSetState` 方法需要完成两件事情：记录新的 state，并把 state 传递到 `setState()` 方法中。这里提供一个示例实现，对 state 进行深复制，并添加到 `this._log` 数组中：

```
var Excel = React.createClass({

  _log: [],

  _logSetState: function(newState) {
    // 克隆一份原来的state并记录下来
    this._log.push(JSON.parse(JSON.stringify(
      this._log.length === 0 ? this.state : newState
    )));
    this.setState(newState);
  },
  // ...

});
```

所有 state 都被记录下来后，还需要一个回放 state 的功能。要触发回放，我们可以简单地添加一个事件监听器，捕捉键盘行为并调用 `_replay()` 函数：

```
componentDidMount: function() {
  document.onkeydown = function(e) {
```

```
    if (e.altKey && e.shiftKey && e.keyCode === 82) { // ALT+SHIFT+R(eplay)
      this._replay();
    }
  }.bind(this);
},
```

最后，我们实现 `_replay()` 方法。它使用 `setInterval()` 方法，每隔一秒钟从记录数组中读取一个 state 对象，并传递给 `setState()`：

```
_replay: function() {
  if (this._log.length === 0) {
    console.warn('No state to replay yet');
    return;
  }
  var idx = -1;
  var interval = setInterval(function() {
    idx++;
    if (idx === this._log.length - 1) { // 结束
      clearInterval(interval);
    }
    this.setState(this._log[idx]);
  }.bind(this), 1000);
},
```

### 3.9.1　如何改进回放功能

撤销 / 重做的功能如何实现？例如，用户按下 ALT+Z 的按键组合时，将 state 记录后退一步，按下 ALT+SHIFT+Z 时则前进一步。

### 3.9.2　有另一种实现方法吗

如果不改变 `setState()` 调用，还有没有另一种方法可以实现类似于回放 / 撤销的功能？也许使用一个合适的生命周期方法（参见第 2 章）可以帮到你？

## 3.10　下载表格数据

实现了排序、编辑、搜索功能以后，用户终于能够获得想要的数据了。如果随后可以让用户把处理好的结果下载下来，不就更完美了吗？

幸运的是，没有什么能难倒 React。你只需要取得当前数据 `this.state.data` 并以 JSON 或者 CSV 格式输出就可以了。

图 3-13 显示了当用户点击 Export CSV 后，下载一个名为 data.csv 的文件（留意浏览器窗口的左下角），然后在 Microsoft Excel 中打开这个表格的最终结果。

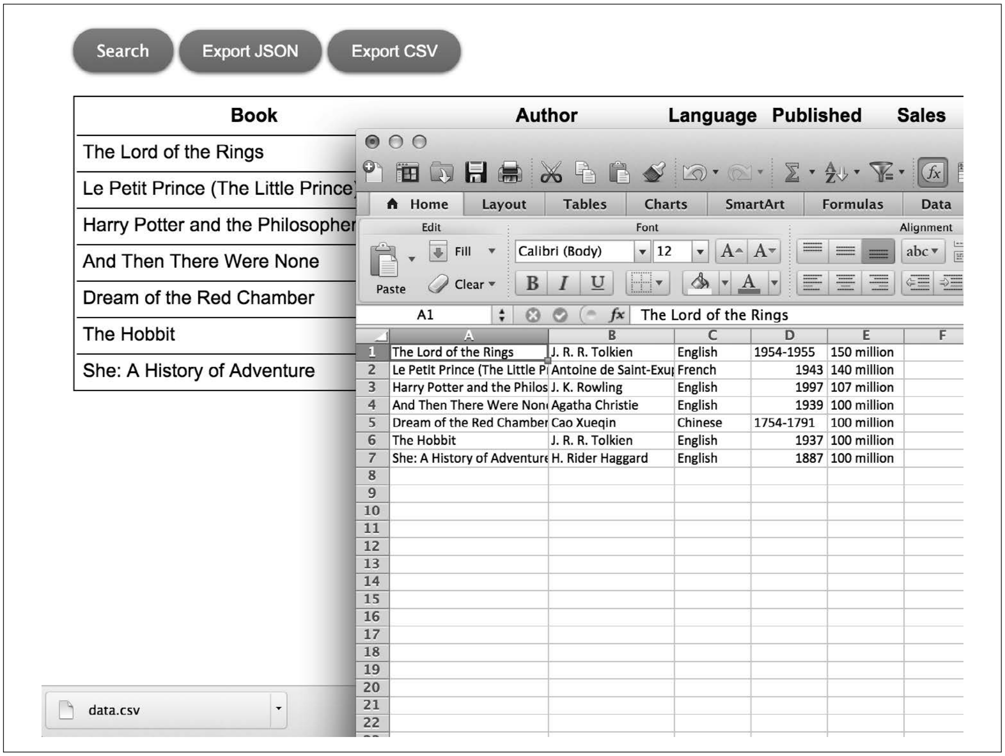



图 3-13：以 CSV 格式输出表格数据到 Microsoft Excel

首先要在工具栏添加一个新的选项。我们使用 HTML5 的特性，在用户点击 <a> 标签时触发文件下载。因此，这两个新的“按钮”其实只不过是用 CSS 修饰的链接而已：

```
_renderToolbar: function() {
  return React.DOM.div({className: 'toolbar'},
    React.DOM.button({
      onClick: this._toggleSearch
    }, 'Search'),
    React.DOM.a({
      onClick: this._download.bind(this, 'json'),
      href: 'data.json'
    }, 'Export JSON'),
    React.DOM.a({
      onClick: this._download.bind(this, 'csv'),
      href: 'data.csv'
    }, 'Export CSV')
  );
},
```

现在实现 _download() 函数。输出 JSON 格式很简单，但输出 CSV 格式需要一些额外步骤。从本质上讲，只需要对所有行和每一行中的所有单元格进行循环，并拼接成一个长字

符串即可。一旦这项工作完成，函数就会通过 `download` 属性和 `window.URL` 创建的 `href` 二进制 blob 来初始化下载功能：

```
_download: function(format, ev) {
  var contents = format === 'json'
    ? JSON.stringify(this.state.data)
    : this.state.data.reduce(function(result, row) {
        return result
          + row.reduce(function(rowresult, cell, idx) {
              return rowresult
                + '"'
                + cell.replace(/"/g, '""')
                + '"'
                + (idx < row.length - 1 ? ',' : '');
            }, '')
          + "\n";
      }, '');

  var URL = window.URL || window.webkitURL;
  var blob = new Blob([contents], {type: 'text/' + format});
  ev.target.href = URL.createObjectURL(blob);
  ev.target.download = 'data.' + format;
},
```

# 第 4 章 JSX

在本书前面的章节中，你已经学会了如何通过 `render()` 方法定义用户界面，以及如何调用 `React.createElement()` 方法和 `React.DOM.*` 函数家族（比如 `React.DOM.span()`）。这种写法有点麻烦，因为过多的函数调用会导致圆括号和花括号的闭合难以跟踪。接下来介绍一种更便捷的写法：JSX。

JSX 是一项独立于 React 且完全可选的技术。如你所见，前面三章甚至没有使用过 JSX。当然，你可以选择不使用 JSX。不过一旦尝试使用这项技术，你就很可能不会回到之前的函数调用方法了。

> JSX 的具体意义很难说清，但通常可以理解为 JavaScriptXML 或者 JavaScript Syntax eXtension（JavaScript 语法扩展）的缩写。JSX 开源项目的官网地址是 http://facebook.github.io/jsx/。

## 4.1 Hello JSX

首先回顾第 1 章的 Hello World 示例代码：

```
<script src="react/build/react.js"></script>
<script src="react/build/react-dom.js"></script>
<script>
  ReactDOM.render(
    React.DOM.h1(
```

```
        {id: "my-heading"},
        React.DOM.span(null,
          React.DOM.em(null, "Hell"),
          "o"
        ),
        " world!"
      ),
      document.getElementById('app')
    );
  </script>
```

这个 `render()` 函数包含好几个函数调用。我们不妨使用 JSX 简化这段代码：

```
ReactDOM.render(
  <h1 id="my-heading">
    <span><em>Hell</em>o</span> world!
  </h1>,
  document.getElementById('app')
);
```

这里的语法和你所熟知的 HTML 非常相似。唯一要注意的地方是，这不是有效的 JavaScript 语法，因此不能在浏览器中直接运行。你需要把这段代码转换（转译，transpile）为浏览器可以正常运行的纯 JavaScript 代码。

## 4.2 转译 JSX

转译的具体过程需要重写源代码，将其转换为老版本浏览器中可以使用的语法。转换后的代码具有和源代码相同的功能。要注意转译与 polyfill 的概念有所不同。

举一个关于 polyfill 的例子，假设我们想要在 `Array.prototype` 中添加一个 `map()` 方法。这个方法是在 ECMAScript5 中引入的，而现在我们要让它在最高支持 ECMAScript3 的浏览器中正常工作：

```
if (!Array.prototype.map) {
  Array.prototype.map = function() {
    // 具体实现
  };
}

// 函数用法
typeof [].map === 'function'; // 返回true,map()方法现在可以使用了
```

polyfill 是纯 JavaScript 领域中的一种解决方案。当你需要为已存在的对象添加新方法或者实现新的对象（比如 `JSON`）时，polyfill 是一种不错的选择。但是当这门语言引入新语法时，polyfill 就不足以帮我们解决问题了。比如，在不支持 `class` 的浏览器中，新的 `class` 关键字是一种无效的语法，浏览器遇到它只能抛出解析异常。在这种情况下没有办法进行 polyfill。因此需要在把代码提供给浏览器使用之前，多加一个编译（转译）新语法的步骤，

以将其转换为浏览器可以正常解析的代码。

转译 JavaScript 正变得越来越普遍，因为程序员希望现在就能使用 ECMAScript6（即 ECMAScript2015）及以后的新特性，而无需等到浏览器支持这些特性。如果你已经设置了一套构建流程（比如进行代码压缩，或者从 ECMAScript6 到 ECMAScript5 的转译操作），那么只需要简单地添加 JSX 转换的步骤就可以了。假设你还不了解构建，接下来我们会简单介绍一些客户端构建的流程。

## 4.3 Babel

Babel（原名 6to5）是一个开源的转译工具，支持转译最新的 JavaScript 特性，也支持转译 JSX。它是你使用 JSX 的前提。在下一章中，你将了解如何设置一个用于生产环境的构建流程，让你可以把 React 应用发布到生产环境。在本章中，我们只对 Babel 作简要介绍，并在客户端完成转译操作。

> 强制性警告：客户端转换仅用作原型展示、教育或探索目的。出于性能考虑，请勿在实际应用中使用该方式进行代码转换。

要在浏览器中（即客户端）进行转换，你需要用到 browser.js 文件。Babel 从版本 6 开始就不再提供这个文件了，但你依然可以从旧版本中找到它：

```
$ mkdir ~/reactbook/babel
$ cd ~/reactbook/babel
$ curl https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.34/browser.js >
browser.js
```

> 在 0.14 版本之前，React 提供了一个客户端 JSX 转换工具。NPM 包 `react-tools` 也提供了一个称为 `jsx` 的命令行工具用于转换。目前，Babel 已经取代了这些工具的作用。

## 4.4 客户端

要让页面中的 JSX 代码正常运行，你需要完成两项工作：

- 引入 browser.js，这个脚本能转译 JSX 代码；
- 在使用 JSX 的 `<script>` 标签中标明 `type` 属性，告诉 Babel 转换这段代码。

目前，在本书的所有例子中，引入 React 库的方式都是这样的：

```
<script src="react/build/react.js"></script>
<script src="react/build/react-dom.js"></script>
```

现在你还需要额外引入转换工具：

```
<script src="react/build/react.js"></script>
<script src="react/build/react-dom.js"></script>
<script src="babel/browser.js"></script>
```

第二步是在 `<script>` 标签中添加 `type` 属性并将其设置为 `text/babel`（浏览器本身不支持该属性），用来告诉 Babel 这段脚本需要进行转换。

之前：

```
<script>
  ReactDOM.render(/*...*/);
</script>
```

之后：

```
<script type="text/babel">
  ReactDOM.render(/*...*/);
</script>
```

当你加载页面时，browser.js 文件会查找所有包含 `text/jsx` 的脚本，并把其中的内容转换为浏览器支持的代码。图 4-1 显示了当你试图在 Chrome 中执行一段 JSX 脚本时发生的情况。正如预料的那样，浏览器会提示语法错误。在图 4-2 中，引入 browser.js 并转译带有 `text/babel` 的脚本块后，页面正常工作。

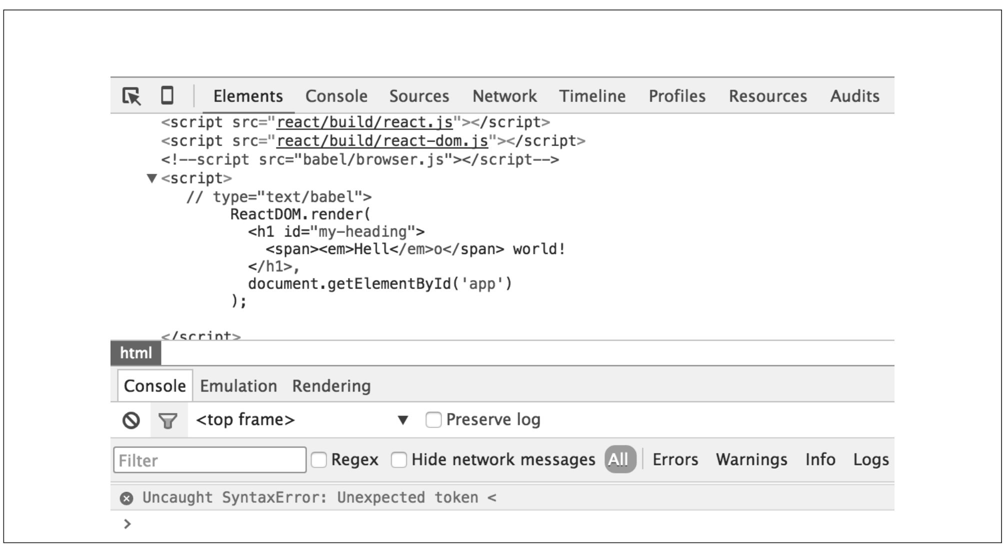



图 4-1：浏览器不能识别 JSX 语法

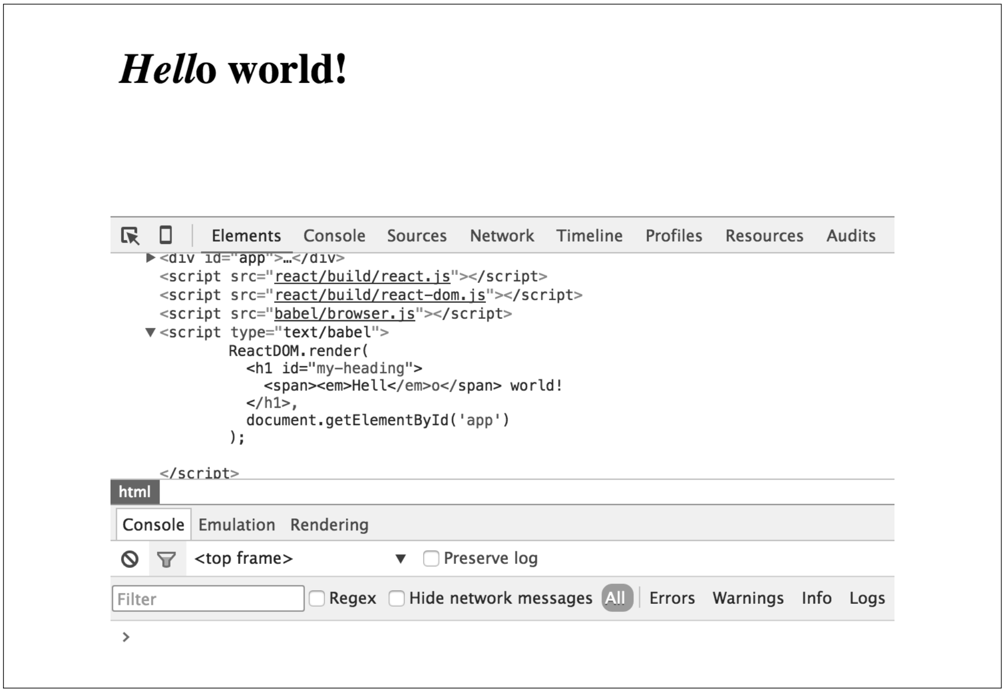



图 4-2：Babel 的浏览器端脚本和类型为 `text/babel` 的脚本

# 4.5 关于 JSX 转换

如果你想尝试并进一步熟悉 JSX 转换，可以使用如图 4-3 所示的在线编辑器（https://babeljs.io/repl/）进行体验。

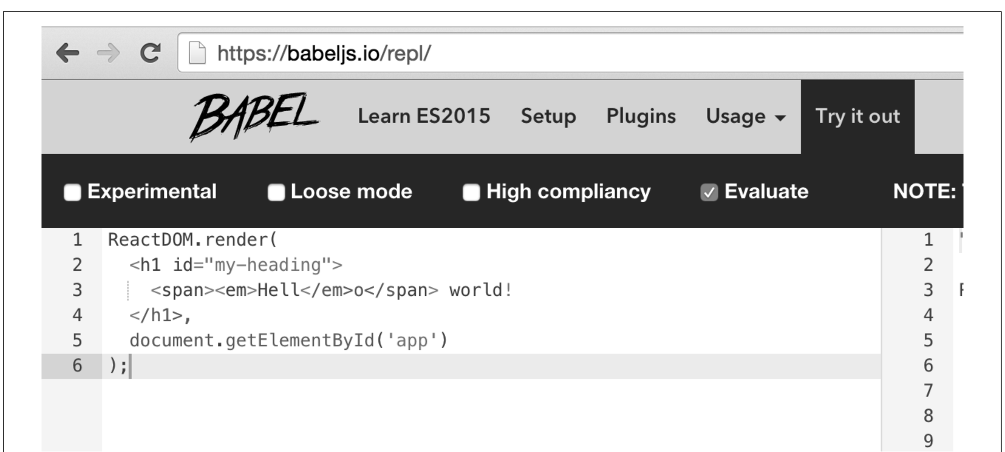



图 4-3：在线 JSX 转换工具

如图 4-4 所示，JSX 转换是轻量级且简单的：JSX 版本的“Hello World”源代码被转换为一系列 `React.createElement()` 调用。这和你之前熟知的函数语法相同。转换结果为纯 JavaScript 代码，因此易于阅读与理解。

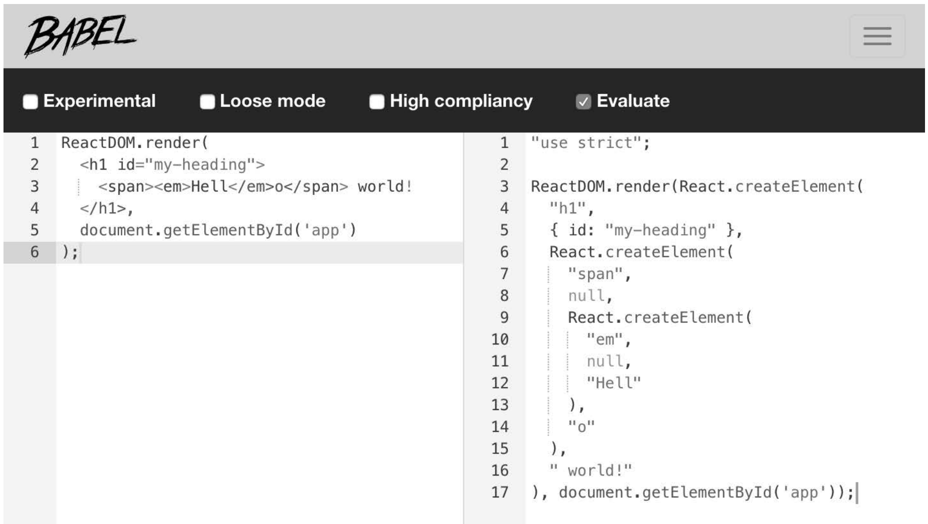



图 4-4：转换后的 Hello World

如果你在学习 JSX 或者想把一份现有的 HTML 代码转为 JSX 语法，还有另一个在线工具可以帮助你：HTML-to-JSX 转换器（https://facebook.github.io/react/html-jsx.html），如图 4-5 所示。

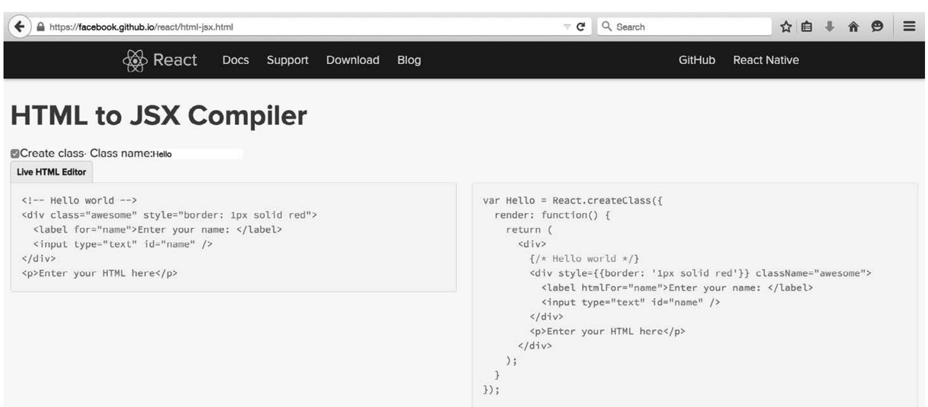



图 4-5：HTML-to-JSX 转换工具

## 4.6 在 JSX 中使用 JavaScript

在构建界面时经常会使用变量、条件判断和循环。JSX 允许你在标记语言中使用 JavaScript 语法，因此不必引入额外的模板语法。在使用时，只需要在 JavaScript 代码外部包裹一层花括号就可以了。

以上一章 `Excel` 组件中的一段代码为例。要把函数语法转换为 JSX，可以这样做：

```
var Excel = React.createClass({

  /* 省略部分代码 */

  render: function() {
    var state = this.state;
    return (
      <table>
        <thead onClick={this._sort}>
          <tr>
            {this.props.headers.map(function(title, idx) {
              if (state.sortby === idx) {
                title += state.descending ? ' \u2191' : ' \u2193';
              }
              return <th key={idx}>{title}</th>;
            })}
          </tr>
        </thead>
        <tbody>
          {state.data.map(function(row, idx) {
            return (
              <tr key={idx}>
                {row.map(function(cell, idx) {
                  return <td key={idx}>{cell}</td>;
                })}
              </tr>
            );
          })}
        </tbody>
      </table>
    );
  }
});
```

如你所见，在使用变量时，需要在外层包裹花括号：

```
<th key={idx}>{title}</th>
```

对于循环体，也需要使用花括号包裹 `map()` 调用：

```
<tr key={idx}>
  {row.map(function(cell, idx) {
    return <td key={idx}>{cell}</td>;
```

```
  })}
</tr>
```

你还可以在 JSX 的 JavaScript 中继续嵌套 JSX，嵌套深度可以根据具体需要而定。不妨把 JSX 看作 JavaScript（经过轻量的转换后）加上熟悉的 HTML 语法。现在你可以让团队中那些不精通 JavaScript 但了解过 HTML 的成员尝试编写 JSX 了。只需要让他们学习一些 JavaScript 基础知识，包括变量、循环等，就可以使用实际数据构建界面了。

在刚才 `Excel` 组件例子的 `map()` 回调函数中，使用了一个 `if` 条件判断。尽管这是一个嵌套的条件判断，但我们可以使用三元表达式加上代码格式化来提高代码的可读性：

```
return (
  <th key={idx}>{
    state.sortby === idx
      ? state.descending
        ? title + ' \u2191'
        : title + ' \u2193'
      : title
  }</th>
);
```

注意到上述例子中重复出现的 `title` 了吗？你可以对代码进行简化：

```
return (
  <th key={idx}>{title}{
    state.sortby === idx
      ? state.descending
        ? ' \u2191'
        : ' \u2193'
      : null
  }</th>
);
```

然而在这种情况下，你还需要修改排序函数。这是因为之前的排序函数假设用户点击了 `<th>` 标签，并通过 `cellIndex` 属性确定用户点击了哪一个 `<th>`。但当你在 JSX 中使用相邻的花括号块时，JSX 会使用 `<span>` 标签把两个花括号中的内容分离开。换句话说，将 `<th>{1}{2}</th>` 转换为 DOM 结构的结果是 `<th><span>1</span><span>2</span></th>`。

## 4.7 在 JSX 中使用空格

在 JSX 中使用空格的方法和 HTML 类似，但又略有不同。举个例子：

```
<h1>
  {1} plus {2} is   {3}
</h1>
```

得到的结果为：

```
<h1>
  <span>1</span><span> plus </span><span>2</span><span> is   </span><span>3</span>
</h1>
```

实际的渲染内容为 1 plus 2 is 3，这与使用 HTML 时的渲染结果相同：多个空格会合并为一个。

然而，在这个例子中：

```
<h1>
  {1}
  plus
  {2}
  is
  {3}
</h1>
```

最终得到的结果为：

```
<h1>
  <span>1</span><span>plus</span><span>2</span><span>is</span><span>3</span>
</h1>
```

如你所见，所有空格都被去除了，输出结果是 1plus2is3。

通过添加 `{'  '}`（这样会产生更多 `<span>` 标签）或者把字符串变为带空格的表达式，可以保证空格总是被添加。换句话说，下列两种方法都是可行的：

```
<h1>
  {/* 空格表达式 */}
  {1}
  {' '}plus{' '}
  {2}
  {' '}is{' '}
  {3}
</h1>

<h1>
  {/* 把空格放在字符串表达式中 */}
  {1}
  {' plus '}
  {2}
  {' is '}
  {3}
</h1>
```

## 4.8 在 JSX 中使用注释

在前面这个例子中，悄悄地使用了一个新概念：在 JSX 标记中添加注释。

由于使用 {} 包裹起来的内容仅仅是 JavaScript，你可以通过 /* 注释内容 */ 的形式添加多行注释。你还可以使用 // 注释内容 添加单行注释，但要注意确保在表达式中的右花括号 } 需要另起一行，否则 } 就会成为注释的一部分：

```
<h1>
  {/* 多行注释 */}
  {/*
    多
    行
    注
    释
    */}
  {
    // 单行注释
  }
  Hello
</h1>
```

由于单行注释 {// 注释 } 不能正常工作（右花括号 } 会被注释掉），使用单行注释并没有什么优势。出于注释风格一致性的考虑，建议在任何情况下都使用多行注释。

## 4.9 HTML 实体

在 JSX 中可以使用 HTML 实体，就像这样：

```
<h2>
  More info &raquo;
</h2>
```

这个例子输出一个右指双尖引号，如图 4-6 所示：




图 4-6：在 JSX 中使用 HTML 实体

然而，如果你在表达式中使用了 HTML 实体，会遇到双重编码的问题。比如在下面这个例子中 HTML 内容会被编码，结果如图 4-7 所示：

```
<h2>
  {"More info &raquo;"}
</h2>
```




图 4-7：双重编码的 HTML 实体

为了避免双重编码的情况，可以使用 Unicode 版本的 HTML 实体作为代替。在这个例子中，右指双尖引号对应的编码是 `\u00bb`（编码表参见 http://dev.w3.org/html5/html-author/charref）：

```
<h2>
  {"More info \u00bb"}
</h2>
```

出于方便起见，你可以在模块顶部定义一个对应该编码的常量。在这里，我们在符号前面再添加一个空格：

```
const RAQUO = ' \u00bb';
```

然后你就可以在任何地方通过常量使用这个符号了：

```
<h2>
  {"More info" + RAQUO}
</h2>

<h2>
  {"More info"}{RAQUO}
</h2>
```

注意到这里使用新的 `const` 关键字代替了 `var` 关键字吗？借助 Babel，你就可以使用 JavaScript 提供的所有最新特性了。关于 Babel 的具体内容将在第 5 章介绍。

## 防范 XSS 攻击

你可能会疑惑：为什么要这样煞费周折地使用 HTML 实体？一个非常重要的目的是防范 XSS 攻击。

为了防范 XSS 攻击，React 会对所有字符串进行转义。当你向用户请求输入某些内容而用户提供了一串恶意的字符串时，React 可以保护你免受攻击。以这种用户输入为例：

```
var firstname = 'John<scr'+'ipt src="http://evil/co.js"></scr'+'ipt>';
```

在某些情况下，你可能需要把这串字符插入 DOM 节点。比如这样：

```
document.write(firstname);
```

然后灾难发生了。页面输出的内容为 John，但随后的 `<script>` 标签却加载了一段恶意 JavaScript 脚本，对你的应用和那些信任你的用户造成了安全危害。

发生这种情况时，React 可以很好地保护你的应用。如果你这样写：

```
React.render(
  <h2>
    Hello {firstname}!
  </h2>,
  document.getElementById('app')
);
```

React 将会对 `firstname` 的内容进行转义（如图 4-8 所示）。

**Hello John<script src="http://evil/co.js"></script>!**

图 4-8：转义字符串

## 4.10 展开属性

JSX 向 ECMAScript6 借鉴了一项实用的特性，称为*展开运算符*（spread operator）。当你定义属性时，可以使用这种便捷的方法。

假设你需要把一系列属性传递给 `<a>` 标签组件：

```
var attr = {
  href: 'http://example.org',
  target: '_blank',
};
```

你当然可以这样写：

```
return (
  <a
    href={attr.href}
    target={attr.target}>
    Hello
  </a>
);
```

但是这种写法过于冗余。使用展开运算符，只需一行代码即可完成相同的功能：

```
return <a {...attr}>Hello</a>;
```

在这个例子中，你（可能根据某些特定条件）提前定义了一个属性对象。这通常是独立使用该组件时的情况；但更常见的使用场景是，从组件外部取得这个对象属性——通常是从父组件传递过来的属性。接下来我们就来看看这种情况。

## 在父子组件之间使用展开属性

假设你正在构建一个 FancyLink 组件，其内部使用的是常规的 <a> 标签实现。你希望组件可以接收所有 <a> 标签可接收的属性（包括 href、style、target 等）以及一些额外的属性（比如 size）。然后其他用户可以这样使用你的组件：

```
<FancyLink
  href="http://example.org"
  style={ {color: "red"} }
  target="_blank"
  size="medium">
  Hello
</FancyLink>
```

这时，你的 render() 函数应该如何利用展开属性的优势，避免重新定义 <a> 标签的所有属性呢？

```
var FancyLink = React.createClass({
  render: function() {
    switch(this.props.size) {
      // 基于size属性进行一些处理
    }

    return <a {...this.props}>{this.props.children}</a>;
  }
});
```

注意到 this.props.children 的使用了吗？这是一个简单方便的方法，允许你传递任意数量的子节点到组件中，并通过这个属性在组合界面时访问其内容。

在前面这个代码片段中，你基于 size 属性进行了一些自定义处理，然后把所有属性简单地传递给 <a> 标签。这当中也包含了 size 属性。React.DOM.a 中没有 size 的概念，因此会自动忽略这个属性，并用上其他所有属性。

你可以对代码进行一些改进，避免传递不必要的属性：

```
var FancyLink = React.createClass({
  render: function() {

    switch(this.props.size) {
      // 基于size属性进行一些处理
```

```
    }

    var attribs = Object.assign({}, this.props); // 浅复制
    delete attribs.size;

    return <a {...attribs}>{this.props.children}</a>;
  }
});
```

使用 ECMAScript7 提议的语法（和前面一样无需额外的工作，Babel 可以进行转换！）可以让代码进一步简化，无需进行对象复制：

```
var FancyLink = React.createClass({
  render: function() {

    var {size, ...attribs} = this.props;

    switch (size) {
      // 基于 size 属性进行一些处理
    }

   return <a{...attribs}>{this.props.children}</a>;
  }
});
```

## 4.11 在 JSX 中返回多个节点

`render()` 函数必须返回单个顶层结点，不允许返回两个顶层结点。换句话说，以下代码会导致错误：

```
// 语法错误：
// 相邻的JSX元素必须包裹在一个闭合标签中

var Example = React.createClass({
  render: function() {
    return (
      <span>
        Hello
      </span>
      <span>
        World
      </span>
    );
  }
});
```

这个问题很容易解决，只要把所有结点包裹在另一个组件中即可，比如 `<div>`：

```
var Example = React.createClass({
  render: function() {
    return (
```

```
      <div>
        <span>
          Hello
        </span>
        <span>
          World
        </span>
      </div>
    );
  }
});
```

虽然不能在 render 方法中返回一个结点数组，但是可以在变量中使用数组，只需要给数组中的结点添加合适的 key 属性即可：

```
var Example = React.createClass({
  render: function() {

    var greeting = [
      <span key="greet">Hello</span>,
      ' ',
      <span key="world">World</span>,
      '!'
    ];

    return (
      <div>
        {greeting}
      </div>
    );
  }
});
```

值得注意的是，你还可以在数组中混入空格与其他字符串，并且不需要给它们添加 key 属性。

在某种程度上，这类似于从父组件接收任意数量的参数，并通过 render() 函数进行传播：

```
var Example = React.createClass({
  render: function() {
    console.log(this.props.children.length); // 4
    return (
      <div>
        {this.props.children}
      </div>
    );
  }
});

React.render(
  <Example>
    <span key="greet">Hello</span>
    {' '}
```

```
    <span key="world">World</span>
    !
  </Example>,
  document.getElementById('app')
);
```

## 4.12　JSX 和 HTML 的区别

JSX 带给你的感觉应该是非常熟悉的，因为它和 HTML 很类似，但 JSX 还支持添加动态值、使用循环与条件判断（只需要使用 {} 包裹起来）。在开始使用 JSX 时，你可以使用 HTML-to-JSX 工具（https://facebook.github.io/react/html-jsx.html），但相信你很快就能学会自己编写 JSX 了。接下来我们会介绍 HTML 和 JSX 之间的一些区别，初学者可能会对此感到吃惊。

在第 1 章中，我们已经讨论过其中一部分区别了，现在来回顾一下。

### 4.12.1　class 和 for 属性不能用了吗

你不能在 JSX 中使用 class 和 for 属性（它们都是 ECMAScript 中的保留字），需要使用 className 和 htmlFor 作为代替：

```
// 错误!
var em = <em class="important" />;
var label = <label for="thatInput" />;

// 正确
var em = <em className="important" />;
var label = <label htmlFor="thatInput" />;
```

### 4.12.2　style 属性值是一个对象

style 属性接收一个对象值，而不是用分号分隔的字符串。CSS 属性名字使用驼峰命名法，而不是使用破折号分隔。

```
// 错误!
var em = <em style="font-size: 2em; line-height: 1.6" />;

// 正确
var styles = {
  fontSize: '2em',
  lineHeight: '1.6'
};
var em = <em style={styles} />;

// 也可以使用内联样式
// 注意双重花括号的使用{ {} }。外层用于包裹JSX的动态值,而内层则代表一个JS对象
var em = <em style={ {fontSize: '2em', lineHeight: '1.6'} } />;
```

### 4.12.3 闭合标签

在 HTML 中，有一些标签不需要闭合；但在 JSX（XML）中，所有标签都要闭合：

```
// 错误!
// 标签没有闭合,虽然在HTML中这样写是合法的
var gimmeabreak = <br>;
var list = <ul><li>item</ul>;
var meta = <meta charset="utf-8">;

// 正确
var gimmeabreak = <br />;
var list = <ul><li>item</li></ul>;
var meta = <meta charSet="utf-8" />;

// 正确
var meta = <meta charSet="utf-8"></meta>;
```

### 4.12.4 用驼峰法命名属性

在之前的代码片段中，你是否注意到了 `charset` 和 `charSet` 的区别？在 JSX 中，所有属性都需要使用驼峰法命名。对于初学者而言，这经常造成混淆——你可能在代码中使用了 `onclick`，但发现它没有达到预期的效果，直到把属性名改为 `onClick` 才能正常工作：

```
// 错误!
var a = <a onclick="reticulateSplines()" />;

// 正确
var a = <a onClick={reticulateSplines} />;
```

需要注意，所有以 `data-` 和 `aria-` 开头的属性都是例外，其命名方式和 HTML 相同。

## 4.13 JSX 和表单

接下来，我们将介绍在处理表单时 JSX 和 HTML 的一些区别。

### 4.13.1 onChange 处理器

在使用表单元素时，用户会与表单进行交互，并改变元素值。在 React 中，你可以通过 `onChange` 属性订阅这些属性发生的更改。该方法可以监听单选按钮和选择框的 `checked` 属性变化，以及 `<select>` 下拉框的 `selected` 属性变化，此外还可以监听文本框和 `<input type="text">` 的输入内容变化，`onChange` 事件会在用户输入时触发，相比于在元素失去焦点时才触发的 `onchange` 原生事件，前者更具实用性。这意味着，我们不需要为了监听用户输入而订阅所有的鼠标事件和键盘事件了。

## 4.13.2　value 和 defaultValue 的区别

在 HTML 中，如果你创建一个输入框 `<input id="i" value="hello" />`，然后在输入框中键入 bye，你会发现：

```
i.value; // "bye"
i.getAttribute('value'); // "hello"
```

而在 React 中，`value` 属性总是和文本输入框的最新内容保持一致。如果你想指定默认值，可以使用 `defaultValue` 属性。

在下面这个代码片段中，你拥有一个 `<input>` 组件。该组件预先填充的内容为 hello，还绑定了一个 `onChange` 处理器。当把 hello 的最后一个字符 o 删掉时，`value` 的值变为 `hell`，而 `defaultValue` 的值仍然是 hello：

```
function log(event) {
  console.log("value: ", event.target.value);
  console.log("defaultValue: ", event.target.defaultValue);
}
React.render(
  <input defaultValue="hello" onChange={log} />,
  document.getElementById('app')
);
```

> 你应当把这种模式应用到自定义组件中：如果你需要接收一个属性，并暗示用户该属性应该一直是最新的（比如 `value`、`data`），就让其保持更新。否则请把属性名改为 `initialData`（在第 3 章讨论过）或者 `defaultValue`，以保证结果符合预期。

## 4.13.3　<textarea> 的值

为了和文本输入框保持一致，React 版本的 `<textarea>` 组件同样能够接收 `value` 和 `defaultValue` 属性，其中 `value` 属性值保持最新，而 `defaultValue` 则和原来的值保持一致。如果你依然坚持 HTML 风格，也就是使用 `<textarea>` 的子节点来定义值（不推荐这样做），React 会把子节点的值定义为 `defaultValue`。

在 HTML 中，`<textarea>`（按照 W3C 的定义）之所以要把子节点作为值，是为了方便开发者在输入内容中进行换行。然而，基于 JavaScript 的 React 不会受到这样的限制。当你需要换行时，只需要使用 `\n` 即可。

请思考下面这个例子的输出结果，答案如图 4-9 所示：

```
function log(event) {
  console.log(event.target.value);
```

```
  console.log(event.target.defaultValue);
}

React.render(
  <textarea defaultValue="hello\nworld" onChange={log} />,
  document.getElementById('app1')
);
React.render(
  <textarea defaultValue={"hello\nworld"} onChange={log} />,
  document.getElementById('app2')
);
React.render(
  <textarea onChange={log}>hello
world
  </textarea>,
  document.getElementById('app3')
);
React.render(
  <textarea onChange={log}>{"hello\n\
world"}
  </textarea>,
  document.getElementById('app4')
);
```




图 4-9：在 `<textarea>` 中换行

要注意文本串 `"hello\nworld"` 与 JavaScript 字符串 `{"hello\nworld"}` 作为属性值时的区别。

此外还要注意，在 JavaScript 中，多行字符串需要使用 \ 进行转义（参见第 4 个例子）。

在这个例子的最后，React 会在使用传统方法设置 `<textarea>` 子节点的值时，在控制台中发出警告。

### 4.13.4 `<select>` 的值

当你在 HTML 中使用 `<select>` 标签时，可以通过 `<option selected>` 指定预先选择的项，

比如这样：

```
<!-- 传统HTML -->
<select>
  <option value="stay">Should I stay</option>
  <option value="move" selected>or should I go</option>
</select>
```

在 React 中，则可以通过 `value` 或者更好的 `defaultValue` 来给 `<select>` 元素指定值：

```
// React/JSX
<select defaultValue="move">
  <option value="stay">Should I stay</option>
  <option value="move">or should I go</option>
</select>
```

这同样适用于多重选择的情况，只需提供一个包含预选择值的数组：

```
<select defaultValue={["stay", "move"]} multiple={true}>
  <option value="stay">Should I stay</option>
  <option value="move">or should I go</option>
  <option value="trouble">If I stay it will be trouble</option>
</select>
```

> 如果你把上述两种方式搞混了，React 会在你为 `<option>` 标签设置 `selected` 属性时给予警告提示。

也可以使用 `<select value>` 代替 `<select defaultValue>`，但是不推荐这样做，因为这需要你手动更新用户看到的值。否则，当用户选择了一个不同的选项时，`<select>` 显示的内容不会发生变化。换而言之，你需要进行类似这样的处理：

```
var MySelect = React.createClass({
  getInitialState: function() {
    return {value: 'move'};
  },
  _onChange: function(event) {
    this.setState({value: event.target.value});
  },
  render: function() {
    return (
      <select value={this.state.value} onChange={this._onChange}>
        <option value="stay">Should I stay</option>
        <option value="move">or should I go</option>
        <option value="trouble">If I stay it will be trouble</option>
      </select>
    );
  }
});
```

## 4.14 使用 JSX 实现 Excel 组件

在本章最后，不妨尝试使用 JSX 重写上一章 `Excel` 组件最终版本中的所有 `render*()` 方法。这项任务是留给你的练习题。如果你感兴趣，可以自己完成，并把答案和本书附带代码库中的示例代码进行对比（https://github.com/stoyan/reactbook/）。

# 第 5 章 为应用开发做准备

除非出于原型展示或测试 JSX 的目的，否则对于任何开发与发布流程，都应该设置构建流程。如果你已经拥有一套现成的构建流程，只需要再添加 Babel 转换的步骤就可以了。假设你还没有设置任何构建流程，我们将从头开始介绍这方面的内容。

我们的目标是无需等待浏览器的原生支持，就能在各种浏览器上使用 JSX 和 JavaScript 的新特性。因此，你需要在开发环境中设置一个转换步骤，并让其在后台运行。这个转换过程生成的代码应当尽量接近用户最终在生产环境中运行的代码。（这意味着不需要再进行客户端转换。）这个过程应当尽量不显眼，以避免在开发和构建的上下文切换中花费大量时间。

当谈及开发与构建过程时，JavaScript 社区和前端生态圈已经提供了大量方案可供选择。我们会把构建过程简单化，不使用任何工具，而是自己动手实现一个构建过程。这样做的好处是：

- 帮助你理解构建过程的原理；
- 让你在随后挑选构建工具时，可以作出更明智的选择；
- 把关注点放在 React 本身，而不是对其他工具的讨论上。

## 5.1 一个模板应用

首先，为新应用构建一个通用的“模板”项目。在这个模板项目中，我们的应用将在客户端运行，并且遵循单页面应用（single-page app，SPA）风格。此外，应用中还会使用 JSX 和 JavaScript 语言本身提供的许多新特性，包括 ES5、ES6（亦称为 ES2015）以及尚在提议中的 ES7 新特性。

### 5.1.1　文件和目录

按照常规做法，你可以建立 /css、/js 和 /images 文件夹用于存放静态资源，然后通过 index.html 文件把静态资源关联起来。接下来，我们将 /js 文件夹进一步划分为 /js/source（使用 JSX 语法编写的脚本）和 /js/build（源代码经过转译后，浏览器可以运行的脚本）。另外，我们还建立了 /scripts 目录，用于存放构建过程用到的命令行脚本。

现在，模板应用的目录结构如图 5-1 所示。

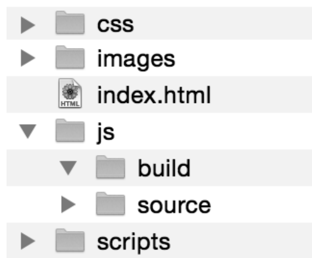



图 5-1：模板应用

下面进一步划分 /css 和 /js 目录（如图 5-2 所示），它们分别包括：

- 在整个应用中通用的文件
- 与某个特定组件相关的文件

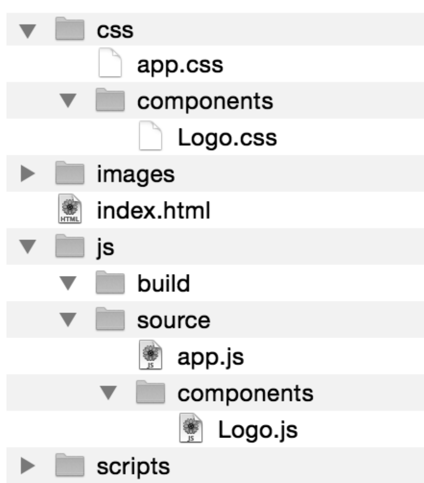



图 5-2：组件的分离

这种划分方式可以帮助你尽可能保持组件的独立性、专用性和可重用性。毕竟，你的目标是使用若干带有特定功能的小组件构建大型应用。这体现了“分而治之”的理念。

最后，我们将尝试创建一个简单的示例组件，并称之为 `<Logo>`（应用一般都拥有标志图案）。一般约定组件的首字母需要大写，因此这里要注意 Logo 和 logo 的区别。为了让组件的相关文件保持命名一致性，我们约定使用 /js/source/components/Component.js 编写组件逻辑，使用 /css/components/Component.css 编写相关样式。图 5-2 显示了完整的文件夹目录结构，其中包含一个简单的 `<Logo>` 组件。

## 5.1.2　index.html

解决了目录结构的问题，接下来看看如何使用这个目录结构编写示例应用。index.html 文件中应该引入如下内容：

- 所有 CSS 文件打包生成的单个 bundle.css 文件；
- 所有 JavaScript 文件打包生成的单个 bundle.js 文件（包括应用中的组件及其依赖库，比如 React）；
- 一如既往，放置应用渲染的容器 `<div id="app">`。

```
<!DOCTYPE html>
<html>
  <head>
    <title>App</title>
    <meta charset="utf-8">
    <link rel="stylesheet" type="text/css" href="bundle.css">
  </head>
  <body>
    <div id="app"></div>
    <script src="bundle.js"></script>
  </body>
</html>
```

单一 .css 和单一 .js 的文件组织形式适用于大部分应用。但当你的应用规模接近 Facebook 和 Twitter 时，初始化加载这些脚本就会非常耗时，而且用户可能不需要一开始就用到所有功能。这时你可以建立一个脚本 / 资源加载器，使得代码可以按需加载。（这个方案将留给你思考。别忘了，有许多开源的解决方案可供选择。）在这种场景下，初始加载的单一 .css 和 .js 文件可以理解为一种引导文件，让用户可以在第一时间看到首屏内容。因此，在应用规模增长时，这种单一文件的模式依然有其用武之地。

你马上就会知道如何把独立的资源文件打包为 bundle.js 和 bundle.css 文件。但在此之前，你需要了解每个 CSS/JS 文件的作用。

### 5.1.3 CSS

全局作用的样式文件 /css/app.css 包含了整个应用的通用样式，如下所示：

```
html {
  background: white;
  font: 16px Arial;
}
```

除全局样式外，你还需要为每个组件定义具体样式。前面我们已经约定每个组件对应一个 CSS 文件（和一个 JavaScript 文件），放置在 /css/components（和 /js/source/components）。现在我们来实现 /css/components/Logo.css 文件：

```
.Logo {
  background-image: url('../../images/react-logo.svg');
  background-size: cover;
  display: inline-block;
  height: 50px;
  vertical-align: middle;
  width: 50px;
}
```

此处还遵循了另一个实用的小约定：在保持组件名称首字母大写的同时，还要给组件的顶层元素设置和该组件名相同的类名，在这里对应为 `className="Logo"`。

### 5.1.4 JavaScript

应用的入口处是 /js/source/app.js 脚本文件。入口文件是所有逻辑开始的地方，因此我们在该文件中这样编写：

```
React.render(
  <h1>
    <Logo /> Welcome to The App!
  </h1>,
  document.getElementById('app')
);
```

最后，在 /js/source/components/Logo.js 文件中，实现示例 React 组件 `<Logo>` 的逻辑：

```
var Logo = React.createClass({
  render: function() {
    return <div className="Logo" />;
  }
});
```

### 5.1.5 更现代化的 JavaScript

直到目前，本书中的例子都只用到了一些简单的组件，而且出于方便起见，让 React 和

ReactDOM 暴露为全局变量。但当你的应用变得复杂，且组件数量越来越多的时候，你就需要使用更好的代码组织形式。这是因为暴露全局变量是有风险的（往往会导致命名冲突），一直依赖全局变量也并不安全。（试想把不同的 JavaScript 打包起来时，内容不是放在单个 bundle.js 中的情形。）

你需要借助模块来解决这个问题。

### 1. 模块化

JavaScript 社区已经提出了好几种模块化方案，其中一种被广泛接受的方案是 CommonJS。在 CommonJS 中，假如你在一个文件中编写了逻辑，就可以导出（export）一个或多个符号（最常见的是对象，不过也可以是函数，甚至单独的变量）：

```
var Logo = React.createClass({/* ... */});

module.exports = Logo;
```

通常约定：一个模块只导出一个内容（比如一个 React 组件）。

现在这个模块需要依赖 React 以调用 React.createClass()。目前没有定义全局变量，因此 React 不能通过全局访问。你需要先进行导入（或者说是 require），就像这样：

```
var React = require('react');

var Logo = React.createClass({/* ... */});

module.exports = Logo;
```

对于接下来开发的每一个组件，我们都将遵循这个模板：在代码顶部声明依赖，在底部导出内容，把组件逻辑放在中间。

### 2. ECMAScript 模块

ECMAScript 规范建议延续了这种模块化思想，并引入了一种新语法（与 require() 和 module.exports 相对）。你可以直接使用这种新语法，Babel 会帮你将其转译为浏览器可以识别的旧语法。

在定义其他模块的依赖关系时，可以把 var React = require('react'); 改为 import React from 'react';。

在导出模块内容时，可以把 module.exports = Logo; 改为 export default Logo。

在 export 语句的末尾省略分号是符合 ECMAScript 语法规范的，并非本书错误。

### 3. 类

现在 ECMAScript 已经引入了类的概念，因此可以使用新的语法。

修改前：

```
var Logo = React.createClass({/* ... */});
```

修改后：

```
class Logo extends React.Component {/* ... */}
```

之前的方法通过一个对象定义 React“类”，其语法和 ECMAScript 2015 中的类语法有一些区别，后者的用法如下。

- 对象中没有定义属性，只有函数（方法）。如果需要定义属性，可以在构造函数中通过 `this` 关键字定义（接下来会介绍更多例子和可选项）。
- 方法通过 `render() {}` 定义，不需要在前面添加 `function` 关键字。
- 方法之间不需要像这样使用逗号分隔：`var obj = {a: 1, b: 2};`。

```
class Logo extends React.Component {
  someMethod() {

  } // 此处不需要添加逗号

  another() { // 此处不需要添加function关键字

  }

  render() {
    return <div className="Logo" />;
  }
}
```

### 4. 概括

随着本书内容的深入，你将接触更多 ECMAScript 的新特性，但目前介绍的语法对于这个模板的开发工作已经足够了，因为模板的作用只是一个最低限度的实现，目的是为新应用的开发建立基础。

现在模板中已经包含了：index.html、全局 CSS 样式（app.css）、每个组件独立的 CSS 样式（/css/components/Logo.css）、JavaScript 代码的入口点（app.js）和按照 React 组件划分的具体模块（比如 /js/source/components/Logo.js）。

以下是 app.js 文件的最终版本：

```
'use strict'; // 总是使用严格模式是一种好习惯

import React from 'react';
```

```
import ReactDOM from 'react-dom';
import Logo from './components/Logo';

ReactDOM.render(
  <h1>
    <Logo /> Welcome to The App!
  </h1>,
  document.getElementById('app')
);
```

以下是 Logo.js 文件的最终版本：

```
import React from 'react';

class Logo extends React.Component {
  render() {
    return <div className="Logo" />;
  }
}

export default Logo
```

你是否注意到了在导入 React 库和导入 Logo 组件时的区别？前者是 `from 'react'`，而后者是 `from './components/Logo'`。后者看起来像一个文件夹路径，而事实也的确如此，模块会从相对路径中导入依赖；而前者则是从一个共享目录（即通过 `npm` 安装的模块目录）中导入依赖。接下来，我们来看看怎么让所有的工作共同起作用，以及新语法是如何在浏览器中完美运行的（甚至包括老版本 IE 浏览器）。

> 你可以在本书附带的代码库中找到这份模板（https://github.com/stoyan/reactbook/），并在此基础上开发你的应用。

## 5.2 安装必备工具

要让 index.html 打开后能显示预期效果，你需要预先完成以下工作。

- 创建 bundle.css。这个文件只是简单地拼接 CSS，因此不需要依赖安装其他工具。
- 让代码在浏览器中可读。你需要使用 Babel 进行转译。
- 创建 bundle.js。我们使用 Browserify 完成这项工作。

Browserify 不仅负责拼接脚本文件，还要完成以下任务。

- 解析并引入所有的依赖。你只需要告诉它 app.js 的位置，它就能找出所有的依赖（包括 React、Logo.js 等）。

- 引入一个 CommonJS 实现，以保证 `require()` 调用可以在浏览器中正常工作。Babel 会把所有的 `import` 语句转化为 `require()` 函数调用。

简而言之：你需要事先安装 Babel 和 Browserify。可以通过 Node.js 附带的 `npm`（node package manager，Node 包管理器）进行安装。

### 5.2.1　Node.js

要安装 Node.js，请打开 http://nodejs.org 并根据你的操作系统类型下载对应的安装程序。下载完成后，跟随指引完成安装。然后你就可以利用 `npm` 安装依赖包了。

要验证是否安装成功，可以在终端中输入：

```
$ npm --version
```

如果你没有使用终端（命令提示符）的经验，现在就是学习的最好时机！如果你使用 Mac OS X，可以点击 Spotlight 搜索（右上角的放大镜图标）并输入 `Terminal`。如果你使用 Windows，找到开始菜单（右键点击屏幕左下方的 Windows 图标），选择“运行”，并输入 `powershell`。

> 在本书中，为了区分终端命令与常规代码，所有在终端中输入的命令都以提示符 `$` 开头。实际在终端中输入命令时，请去掉 `$` 符号。

### 5.2.2　Browserify

在终端中输入如下命令，可以通过 `npm` 安装 Browserify：

```
$ npm install --global browserify
```

要验证是否安装成功，输入：

```
$ browserify --version
```

### 5.2.3　Babel

要安装 Babel 的命令行界面（command-line interface，CLI），输入：

```
$ npm install --global babel-cli
```

要验证是否安装成功，输入：

```
$ babel --version
```

发现规律了吗？

> 通常情况下，推荐在项目本地安装 Node 包，也就是去掉上述例子中的 `--global` 标记（参见另一种模式：`global === bad?`）。在本地安装依赖包时，你可以安装同一个包的不同版本，以满足你的应用以及引入库的依赖关系。但对于 Browserify 和 Babel 来说，在全局范围安装能方便你在全局范围（任何一个目录）通过命令行界面进行访问。

### 5.2.4 React 相关

最后还需要安装几个 React 相关的依赖包：

- 首先当然是 `react`；
- `react-dom`，它是独立于 React 的；
- `babel-preset-react`，让 Babel 支持 JSX 以及其他 React 语法；
- `babel-preset-es2015`，提供了对新版本 JavaScript 特性的支持。

首先进入应用目录（输入 `cd ~/reactbook/reactbook-boiler` 命令），然后就可以在本地安装这些包了：

```
$ npm install --save-dev react
$ npm install --save-dev react-dom
$ npm install --save-dev babel-preset-react
$ npm install --save-dev babel-preset-es2015
```

接下来你会注意到，应用目录中出现了一个 node_modules 目录，其中包含本地安装的包及其依赖包。前面两个全局安装的模块（Babel 和 Browserify）则放在了另一个 node_modules 目录中，其具体位置和操作系统有关（比如 /usr/local/lib/node_modules 或者 C:\Users{ 用户名 }\AppData\Roaming\npm\）。

## 5.3 开始构建

构建过程需要完成三件事情：CSS 拼接、JavaScript 转译和 JavaScript 打包。这三个过程都很简单，只需运行三条命令即可。

### 5.3.1 转译 JavaScript

首先通过 Babel 转译 JavaScript：

```
$ babel --presets react,es2015 js/source -d js/build
```

这条命令从 js/source 文件夹中读取所有文件，并转译其中的 React 和 ES2015 语法，并把结果复制到 js/build 中。你会在命令行中看到类似这样的输出结果：

```
js/source/app.js -> js/build/app.js
js/source/components/Logo.js -> js/build/components/Logo.js
```

这个列表内容会随着你的组件内容增多而增加。

### 5.3.2 打包 JavaScript

接下来进行打包：

```
$ browserify js/build/app.js -o bundle.js
```

你告诉 Browserify：应用入口为 js/build/app.js，找出其中所有依赖并把结果输出到文件 bundle.js 中。最后你需要在 index.html 的结尾处引入这个文件。要检查输出文件的内容，可以输入 less bundle.js。

### 5.3.3 打包 CSS

CSS 打包非常简单（至少在现阶段是如此），你甚至不需要借助特别的工具来完成，只需要把所有的 CSS 文件拼接成一个就可以了（使用 `cat` 命令）。可是，由于移动了文件路径，CSS 中原有的图像路径会失效，因此我们还需要使用 `sed` 命令简单地进行替换：

```
cat css/*/* css/*.css | sed 's/..\/..\/images/images/g' > bundle.css
```

有一些 NPM 包可以帮助你更好地完成这些工作，但目前我们还不需要它们。

### 5.3.4 大功告成

现在你已经完成了构建过程，可以准备查看辛勤劳动后的成果了。在浏览器中打开 index.html 文件，你将会看见如图 5-3 所示的欢迎界面。

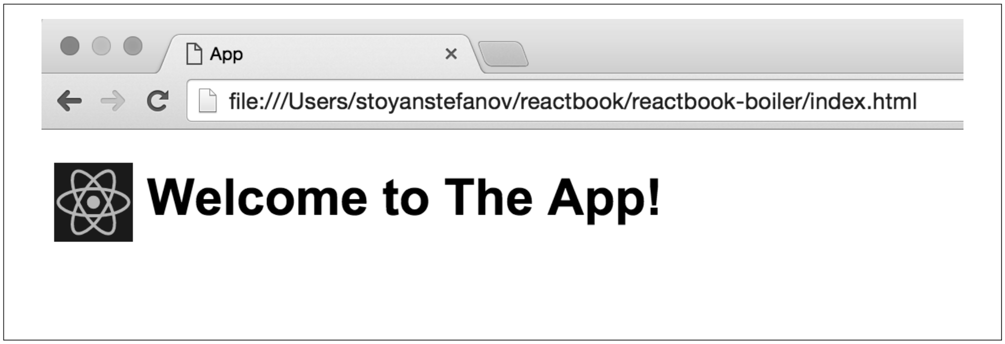



图 5-3：欢迎使用应用

## 5.3.5　Windows 版本

上述命令只适合在 Linux 和 Mac OS X 系统中使用。不过在 Windows 系统上也没有太大区别。对于前两条命令，除了目录分隔符外，其他地方都是相同的：

```
$ babel --presets react,es2015 js\source -d js\build
$ browserify js\build\app.js -o bundle.js
```

在 Windows 系统上没有 cat 命令，但是你可以这样拼接文件：

```
$ type css\components\* css\* > bundle.css
```

要替换字符串（让 CSS 在 images 中寻找图片，而不是 ../../images），你需要借助 powershell 的一些高端特性：

```
$ (Get-Content bundle.css).replace('../../images', 'images') | Set-Content bun
dle.css
```

## 5.3.6　在开发过程中构建

每次修改文件后都要手动运行构建过程是一件痛苦的事情。幸好，你可以通过脚本监听目录中的文件改变，并自动运行构建脚本。

首先，我们把构建过程用到的三条命令放到一个文件中，并命名为 scripts/build.sh：

```
# 转换js
babel --presets react,es2015 js/source -d js/build
# 打包js
browserify js/build/app.js -o bundle.js
# 打包css
cat css/*/* css/*.css | sed 's/..\/..\/images/images/g' > bundle.css
# 完成
date; echo;
```

接下来，安装一个名为 `watch` 的 NPM 包：

```
$ npm install --save-dev watch
```

运行 `watch` 命令对 js/source/ 和 /css 目录中的任意更改进行监听，一旦文件内容发生变化，就运行 scripts/build.sh 文件中的脚本：

```
$ watch "sh scripts/build.sh" js/source css

> Watching js/source/
> Watching css/
js/source/app.js -> js/build/app.js
js/source/components/Logo.js -> js/build/components/Logo.js
Sat Jan 23 19:41:38 PST 2016
```

当然，你也可以把命令放在 scripts/watch.sh 中。每当你进行应用开发时，只需要运行如下代码即可：

```
$ sh scripts/watch.sh
```

你可以对源文件进行修改，然后刷新浏览器来查看新的构建。

## 5.4 发布

目前差不多可以发布应用了，因为你在开发过程中已经进行了构建，所以发布过程没有太大的工作量。不过在应用正式上线之前，你可能还需要作一些额外的处理，比如代码压缩和图像优化。

我们以常用的 JavaScript 压缩工具 `uglify` 和 CSS 压缩工具 `cssshrink` 为例，实现一套简单的发布流程。你可以在此基础上压缩 HTML 代码、优化图像、复制文件到内容分发网络（content delivery network，CDN），做其他任何你需要的事情。

scripts/deploy.sh 文件的内容如下：

```
# 删除上一个版本
rm -rf __deployme
mkdir __deployme

# 构建
sh scripts/build.sh

# 压缩JavaScript
uglify -s bundle.js -o __deployme/bundle.js
# 压缩CSS
cssshrink bundle.css > __deployme/bundle.css
# 复制HTML和图片
cp index.html __deployme/index.html
```

```
cp -r images/ __deployme/images/

# 完成
date; echo;
```

在脚本运行完毕后，你会得到一个新的目录。这个名为 __deployme 的目录中包含以下内容：

- index.html
- 压缩后的 bundle.css
- 压缩后的 bundle.js
- images/ 文件夹

接下来你只需要把整个目录复制到服务器上，就可以为用户提供这个新版本的应用了。

## 5.5 更进一步

现在你已经拥有了一个简单的基于 shell 脚本的构建和发布流程。你可以根据具体需要扩展这些脚本，也可以尝试使用一些专业的构建工具（比如 Grunt 或者 Gulp）进行构建，以便更好地满足你的需求。

在完成所有的构建和转译流程后，我们将关注一个更有趣的话题：利用 JavaScript 提供的各种新特性，构建并测试一款真正的应用。

第 6 章

# 构建应用

到目前为止，你已经学会：创建 React 自定义组件（以及使用内建组件）的所有相关基础知识，使用 JSX 定义用户界面（可选）的方法，以及构建并发布应用的流程。是时候创建一个更完整的应用了。

我们把这个应用称为 Whinepad，用户可以在这个应用中对品尝过的酒类写下笔记并进行评价。事实上，用户不仅可以评价酒类，还可以留下任何想要*抱怨*[1] 的内容。这个应用应该涵盖常见的添加、查询、修改、删除（CRUD）功能。此外，它还是一个把数据存储在客户端的纯客户端应用。由于本书以学习 React 为目的，应用中所有与 React 不相关的部分（比如数据存储、样式等）将不作详述。

在本章中，你将学习以下内容：

- 从可重用的小组件开始，构建整个应用；
- 进行组件间通信，让不同的组件共同发挥作用。

## 6.1 Whinepad v.0.0.1

我们将在上一章建立的模板应用的基础上进行 Whinepad 应用的开发。当你品尝了一些新的酒类之后，可以在这个应用中记录笔记并进行评价。不如就把欢迎界面设置为曾经评价过的内容列表吧？这只需简单地重用第 3 章创建的 `<Excel>` 组件即可。

注 1：英文原文是 whine，与酒类（wine）同音。这个单词在应用名中用作双关。——编者注

### 6.1.1 基本设置

首先复制一份 reactbook-boiler 模板应用的源代码，然后把项目重命名为 whinepad v0.0.1，并在此基础上进行开发。（可以从 https://github.com/stoyan/reactbook/ 下载代码。）下一步，运行监听脚本，以便在文件内容发生改变时自动重新构建：

```
$ cd ~/reactbook/whinepad\ v0.0.1/
$ sh scripts/watch.sh
```

### 6.1.2 开始编写代码

修改 index.html 文件中的标题，并把 id 属性修改为 id="pad"，以匹配我们的应用名称：

```
<!DOCTYPE html>
<html>
  <head>
    <title>Whinepad v.0.0.1</title>
    <meta charset="utf-8">
    <link rel="stylesheet" type="text/css" href="bundle.css">
  </head>
  <body>
    <div id="pad"></div>
    <script src="bundle.js"></script>
  </body>
</html>
```

我们把 JSX 版本 Excel 组件的源代码（出现在第 4 章末尾处）复制到 js/source/components/Excel.js 中：

```
import React from 'react';

var Excel = React.createClass({

  // 省略部分代码

  render: function() {
    return (
      <div className="Excel">
        {this._renderToolbar()}
        {this._renderTable()}
      </div>
    );
  },

  // 省略部分代码
});

export default Excel
```

上述代码和之前的 Excel 组件代码有一些不同，在于：

- 使用了 import/export 语句；
- 遵循上一章的约定，在组件的顶层标签添加了 className="Excel" 属性。

相应地，所有 CSS 样式需要添加前缀，就像这样：

```
.Excel table {
  border: 1px solid black;
  margin: 20px;
}

.Excel th {
  /* 样式 */
}

/* 更多样式 */
```

目前还剩下一项工作，就是在主文件 app.js 中引入 <Excel> 组件。前面已经提到，我们将在客户端存储数据（localStorage）。在起步阶段，为了避免页面空白，我们构造一些初始数据：

```
var headers = localStorage.getItem('headers');
var data = localStorage.getItem('data');

if (!headers) {
  headers = ['Title', 'Year', 'Rating', 'Comments'];
  data = [['Test', '2015', '3', 'meh']];
}
```

接下来把数据传递到 <Excel> 组件中：

```
ReactDOM.render(
  <div>
    <h1>
      <Logo /> Welcome to Whinepad!
    </h1>
    <Excel headers={headers} initialData={data} />
  </div>,
  document.getElementById('pad')
);
```

在 Logo.css 文件中再增添一些样式，你就完成了这个 0.0.1 版本（如图 6-1 所示）！

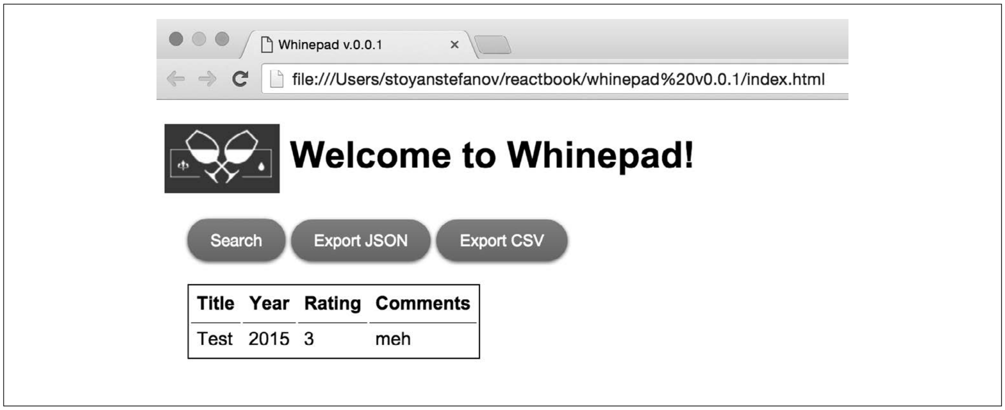



图 6-1：Whinepad v.0.0.1

## 6.2　组件

在开始阶段，我们轻松地重用了现有的 `<Excel>` 组件；然而这个组件已经包含了太多内容。为了体现“分而治之”的思想，我们最好把组件细分为更小、更可重用的组件。举个例子，按钮就应该是一个独立的组件，方便我们在 `<Excel>` 表格以外的地方使用。

此外，这个应用还需要一些专用的组件。比如评分组件，让我们可以通过星星显示打分，而非仅仅显示一个数字。

现在我们来配置这个新的应用，并添加一个辅助工具——组件发现工具。这个工具可以帮你做到下面两点。

- 在一个隔离的环境中开发并测试组件。通常情况下，在应用中开发组件会导致组件和应用的强耦合，从而降低组件的可用性。独立地开发组件可以帮助你在思考如何解耦时作出更明智的选择。
- 方便团队中的其他成员发现并重用已有的组件。随着应用规模的增长，团队成员也会增多。为了避免在同一个组件上出现多人重复开发、导致人力浪费的情况，并且为了促进组件的可重用性（有助于提高应用开发效率），推荐做法是把所有组件以及如何使用的例子全部放在一个地方。

### 6.2.1　设置

先按下 CTRL+C 结束旧的监听脚本进程，以便开始一个新的。把初始的最小可行产品（minimum viable product，MVP）`whinepad v.0.0.1` 复制到一个名为 whinepad 的新目录中：

```
$ cp -r ~/reactbook/whinepad\ v0.0.1/ ~/reactbook/whinepad
$ cd ~/reactbook/whinepad
```

```
$ sh scripts/watch.sh

> Watching js/source/
> Watching css/
js/source/app.js -> js/build/app.js
js/source/components/Excel.js -> js/build/components/Excel.js
js/source/components/Logo.js -> js/build/components/Logo.js
Sun Jan 24 11:10:17 PST 2016
```

### 6.2.2 组件发现工具

我们把这个组件发现工具命名为 discovery.html，并放置在根目录下：

```
$ cp index.html discovery.html
```

这个工具不需要加载整个应用，所以我们不必加载 app.js，只需要加载 discover.js 就可以了，后者包含了所有的组件示例。因此，你也不必引入应用的 bundle.js，只需引入一个单独的包，比如 discover-bundle.js：

```
<!DOCTYPE html>
<html>
  <!-- 和index.html一样 -->
  <body>
    <div id="pad"></div>
    <script src="discover-bundle.js"></script>
  </body>
</html>
```

额外进行一次打包也很简单，只需要在 build.sh 脚本中添加一行：

```
# js打包
browserify js/build/app.js -o bundle.js
browserify js/build/discover.js -o discover-bundle.js
```

最后，把示例组件 <Logo> 添加到发现工具中（js/source/discover.js）：

```
'use strict';

import Logo from './components/Logo';
import React from 'react';
import ReactDOM from 'react-dom';

ReactDOM.render(
  <div style={ {padding: '20px'} }>
    <h1>Component discoverer</h1>

    <h2>Logo</h2>
    <div style={ {display: 'inline-block', background: 'purple'} }>
      <Logo />
    </div>
```

```
    {/* 可以在此放置更多的组件示例 */}

  </div>,
  document.getElementById('pad')
);
```

每当你编写了新的组件后，就可以在这个新组件发现工具（如图 6-2 所示）中进行体验。我们需要在每个新组件诞生后，及时构建并更新这个工具。

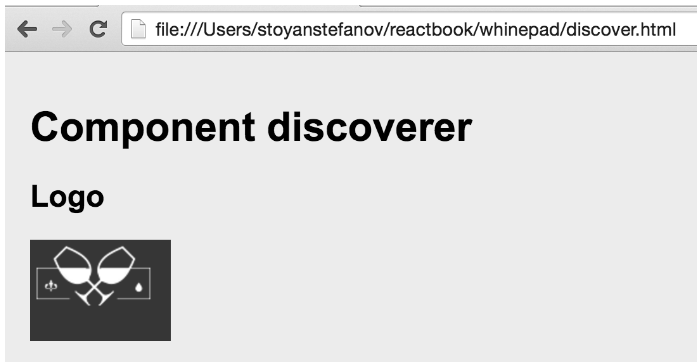



图 6-2：Whinepad 的组件发现工具

## 6.2.3 <Button> 组件

毫不夸张地说，每个应用都离不开按钮组件。我们通常使用原生的 `<button>` 标签按钮，但有时候可能不得不使用 `<a>` 标签按钮，在第 3 章中用到的下载按钮就属于后者。让我们创建一个更通用的 `<Button>` 按钮组件，接收一个可选的 `href` 属性，以应对这两种情况吧。如果 `href` 属性存在，则渲染 `<a>` 标签按钮。

本着测试驱动开发（test-driven development，TDD）的思想，你可以先在 discovery.js 工具中定义这个组件的示例用法。

修改前：

```
import Logo from './components/Logo';

{/* ... */}

{/* 可以在这里放置更多的组件示例 */}
```

修改后：

```
import Button from './components/Button';
import Logo from './components/Logo';
```

```
{/* ... */}

<h2>Buttons</h2>
<div>Button with onClick: <Button onClick={() => alert('ouch')}>Click me</Button></
div>
<div>A link: <Button href="http://reactjs.com">Follow me</Button></div>
<div>Custom class name: <Button className="custom">I do nothing</Button></div>

{/* 可以在这里放置更多的组件示例 */}
```

［或许我们可以把这种方式称为发现驱动开发（discovery-driven development，DDD）？］

> 注意到上述代码中的 `() => alert('ouch')` 模式了吗？这是 ES2015 中箭头函数的一种示例用法。
>
> 以下是箭头函数的其他用法。
>
> - `() => {}`　这是一个空函数，和 `function() {}` 类似。
> - `(what, not) => console.log(what, not)`　这是一个带参数的函数。
> - `(a, b) => { var c = a + b; return c;}`　当函数体中包含多个语句的时候，需要使用花括号 `{}` 包裹。
> - `let fn = arg => {}`　当函数只接收一个参数时，可以省略圆括号 `()`。

## 6.2.4　Button.css

根据之前的约定，`<Button>` 组件的样式应该放在 /css/components/Button.css 文件中。这个文件没有什么特别的地方，只包含了一些用于美化的 CSS 样式。这里仅列出一套样式作为示范，接下来将不再花费时间详述其他组件的 CSS 样式：

```
.Button {
  background-color: #6f001b;
  border-radius: 28px;
  border: 0;
  box-shadow: 0px 1px 1px #d9d9d9;
  color: #fff;
  cursor: pointer;
  display: inline-block;
  font-size: 18px;
  font-weight: bold;
  padding: 5px 15px;
  text-decoration: none;
  transition-duration: 0.1s;
  transition-property: transform;
}

.Button:hover {
  transform: scale(1.1);
}
```

### 6.2.5 Button.js

首先完整阅读一遍 /js/source/components/Button.js 的源代码：

```
import classNames from 'classnames';
import React, {PropTypes} from 'react';

function Button(props) {
  const cssclasses = classNames('Button', props.className);
  return props.href
    ? <a {...props} className={cssclasses} />
    : <button {...props} className={cssclasses} />;
}

Button.propTypes = {
  href: PropTypes.string,
};

export default Button
```

这个组件虽然代码量不多，但是代码中出现了许多新的概念和语法。让我们从头开始分析这段代码！

#### 1. classnames 包

```
import classNames from 'classnames';
```

（通过 `npm i --save-dev classnames` 安装的）`classnames` 包提供了一个处理 CSS 类名的辅助函数。我们经常会碰到这种情况：要让组件既拥有自身的类名，又能从父组件灵活地接收自定义的类名。在过去，React 的插件包可以完成这项任务，但现在我们更推荐使用 `classnames` 这个第三方包来完成。`classnames` 的使用方式很简单，只需要一个函数：

```
const cssclasses = classNames('Button', props.className);
```

在创建组件时，这行代码负责把类名 `Button` 和传递进来的任意类名（如果有的话）合并起来（如图 6-3 所示）。

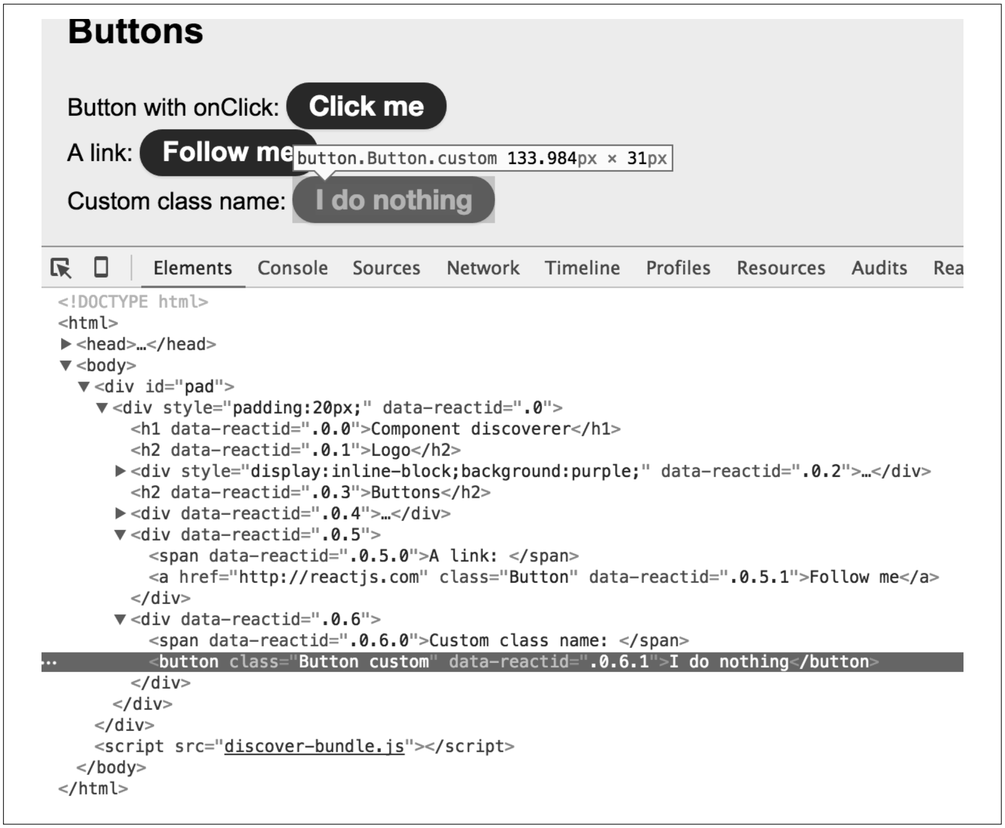



图 6-3：带有自定义类名的 <Button> 组件

你也可以选择不依赖第三方包，自己进行类名的拼接，但 classnames 对其稍微进行了封装，让你可以更加方便地完成这项工作。此外，它还让你可以根据条件有选择地设定类名，用法也很方便，比如：

```
<div className={classNames({
  'mine': true, // 总是包含该类名
  'highlighted': this.state.active, // 根据组件的 state 决定
  'hidden': this.props.hide, // 也可以根据 props 决定
})} />
```

### 2. 解构赋值

```
import React, {PropTypes} from 'react';
```

上述语句是以下声明方法的简写形式：

```
import React from 'react';

const PropTypes = React.PropTypes;
```

### 3. 无状态函数式组件

当一个组件的功能非常简单（这没有任何不妥之处）且不需要维护状态时，可以选择使用函数定义这个组件。函数体的内容等价于 `render()` 方法中的内容。该函数接收的第一个参数包含了所有属性——这就是在函数体中使用 `props.href`，而非使用类或对象中 `this.props.href` 的原因。

可以使用箭头函数对函数进行重写：

```
const Button = props => {
  // ...
};
```

还可以把函数体简化为一句话，进而省略 `{}`、`;` 和 `return`：

```
const Button = props =>
  props.href
    ? <a {...props} className={classNames('Button', props.className)} />
    : <button {...props} className={classNames('Button', props.className)} />
```

### 4. propTypes

如果你使用了 ES2015 的类或者函数式组件的语法，你需要在组件定义后，把类似 `propTypes` 这样的属性以静态属性的方式进行定义。

旧版语法（ES3、ES5）：

```
var PropTypes = React.PropTypes;

var Button = React.createClass({
  propTypes: {
    href: PropTypes.string
  },
  render: function() {
    /* 渲染 */
  }
});
```

新版语法（ES2015 的类）：

```
import React, {Component, PropTypes} from 'react';

class Button extends Component {
  render() {
    /* 渲染 */
  }
}

Button.propTypes = {
  href: PropTypes.string,
};
```

如果使用无状态函数式组件，用法如下：

```
import React, {Component, PropTypes} from 'react';

const Button = props => {
  /* 渲染 */
};

Button.propTypes = {
  href: PropTypes.string,
};
```

## 6.2.6 表单

我们目前已经完成了 `<Button>` 组件的创建。接下来进行一项对于任何数据录入应用都必不可少的工作：处理表单。作为应用开发者，我们很少满足于浏览器内建表单的样式和使用体验，因此更倾向于创建自定义版本的表单。这个 Whinepad 应用也不例外。

我们将创建一个通用的 `<FormInput>` 组件，其中包含一个让调用者获取用户输入内容的 `getValue()` 方法。根据所传入 `type` 属性值的不同，组件负责把输入元素的创建工作委托给具体的组件进行创建，比如 `<Suggest>` 组件、`<Rating>` 等。

我们先从相对底层的组件开始，它们都只需要实现自身的 `render()` 和 `getValue()` 方法。

## 6.2.7 `<Suggest>`

在 Web 开发中，经常可以见到各式各样带有自动提示功能（又名 typeahead）的输入框，其中有些相当出色，但我们打算直接借助浏览器已经提供的 `<datalist>`HTML 元素（https://developer.mozilla.org/en/docs/Web/HTML/Element/datalist）把这项功能做得简单一点（如图 6-4 所示）。

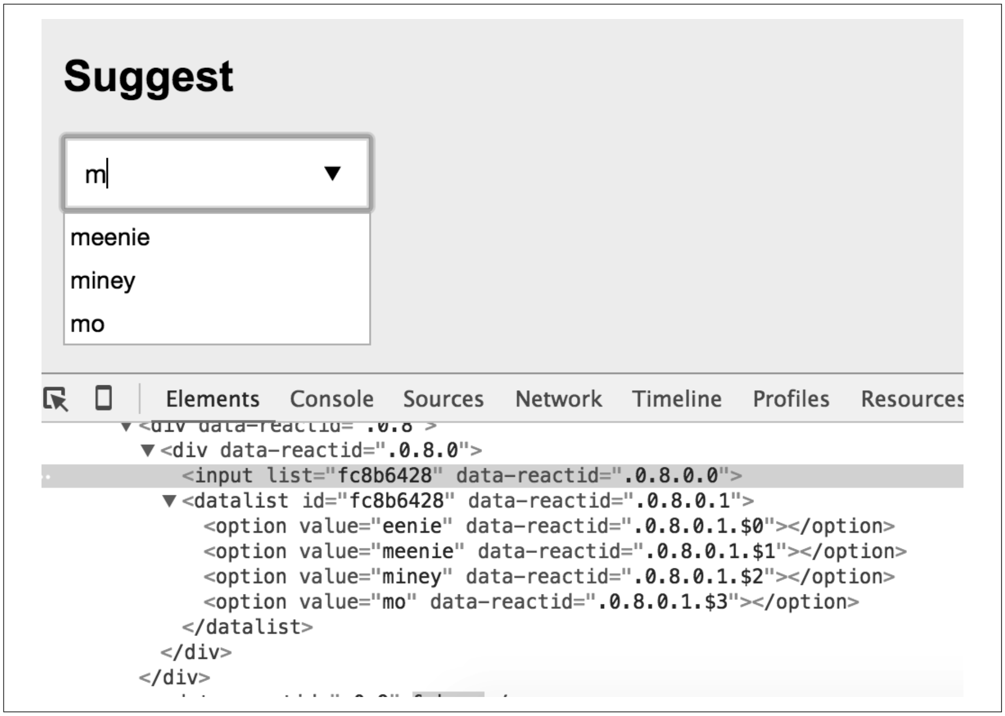



图 6-4：`<Suggest>` 输入框

首要任务是更新发现工具：

```
<h2>Suggest</h2>
<div><Suggest options={['eenie', 'meenie', 'miney', 'mo']} /></div>
```

接下来在 /js/source/components/Suggest.js 文件中实现组件逻辑：

```
import React, {Component, PropTypes} from 'react';

class Suggest extends Component {

  getValue() {
    return this.refs.lowlevelinput.value;
  }

  render() {
    const randomid = Math.random().toString(16).substring(2);
    return (
      <div>
        <input
          list={randomid}
          defaultValue={this.props.defaultValue}
          ref="lowlevelinput"
          id={this.props.id} />
```

```
        <datalist id={randomid}>{
          this.props.options.map((item, idx) =>
            <option value={item} key={idx} />
          )
        }</datalist>
      </div>
    );
  }
}

Suggest.propTypes = {
  options: PropTypes.arrayOf(PropTypes.string),
};

export default Suggest
```

上述的组件代码没有什么特别的地方，只是简单地封装了一个 `<input>` 并（通过 `randomid`）附带了一个 `<datalist>`。

根据 ES 的新语法，还可以利用*解构赋值*（destructuring assignment）从对象中提取多个属性并赋值给同一个变量：

```
// 旧语法
import React from 'react';
const Component = React.Component;
const PropTypes = React.PropTypes;

// 新语法
import React, {Component, PropTypes} from 'react';
```

至于 React 的新概念，这里用到了 `ref` 属性。

### ref

思考如下代码：

```
<input ref="domelement" id="hello">
/* 随后 */
console.log(this.refs.domelement.id === 'hello'); // 输出true
```

你可以利用 `ref` 属性对特定的 React 组件实例进行命名，并在随后*引用*该组件。你可以为任何组件添加 `ref` 属性，但通常是当你需要取得组件的底层 DOM 元素时才需要这么做。使用 `ref` 是一种变通的方法，通常应该还有其他方法可以实现相同的功能。

在上面的例子中，你希望在需要时取得 `<input>` 的值。我们还可以将输入框内容的改变理解为组件 state 发生了改变，因此可以改为使用 `this.state` 进行跟踪：

```
class Suggest extends Component {

  constructor(props) {
```

```
    super(props);
    this.state = {value: props.defaultValue};
  }

  getValue() {
    return this.state.value; // 不需要用ref了
  }

  render() {}
}
```

这样 `<input>` 不再需要添加 `ref` 属性了，但是需要添加 `onChange` 事件监听器，以便更新状态：

```
<input
  list={randomid}
  defaultValue={this.props.defaultValue}
  onChange={e => this.setState({value: e.target.value})}
  id={this.props.id} />
```

注意在构造函数 `constructor()` 中使用了 `this.state = {};`。这种用法在 ES6 中替代了原本的 `getInitialState()` 函数。

## 6.2.8 <Rating> 组件

这个应用是用于对已品尝酒类记录笔记的。最偷懒的一种记录方法就是进行星级评分，比如从一星到五星。

通过一些可配置的选项，可以让这个星级评分组件变为高度可重用。

- 显示的星级数量可以为任意数值。默认值是 5，但是也可以变为其他数值，比如 11。
- 只读属性。有些时候你不愿让重要的评分数据因为用户误点击而丢失。

首先在发现工具中编写测试（如图 6-5 所示）：

```
<h2>Rating</h2>
<div>No initial value: <Rating /></div>
<div>Initial value 4: <Rating defaultValue={4} /></div>
<div>This one goes to 11: <Rating max={11} /></div>
<div>Read-only: <Rating readonly={true} defaultValue={3} /></div>
```




图 6-5：星级评分组件

在编写具体逻辑之前，需要设置属性类型，并列出需要维护的状态：

```
import classNames from 'classnames';
import React, {Component, PropTypes} from 'react';

class Rating extends Component {

  constructor(props) {
    super(props);
    this.state = {
      rating: props.defaultValue,
      tmpRating: props.defaultValue,
    };
  }

  /* 更多方法 */

}

Rating.propTypes = {
  defaultValue: PropTypes.number,
  readonly: PropTypes.bool,
  max: PropTypes.number,
};

Rating.defaultProps = {
  defaultValue: 0,
  max: 5,
};

export default Rating
```

这些属性的作用一目了然：`max` 代表星星的数量，`readonly` 决定组件是否只读。在 state 中，`rating` 属性对应当前的评分，`tmpRating` 则对应用户把鼠标放在星星上移动但尚未点击提交时显示的评分。

接下来编写一些辅助函数，帮助我们处理用户与组件进行交互时发生的状态变化：

```
getValue() { // 我们的所有输入组件都提供了这个函数
  return this.state.rating;
}

setTemp(rating) { // 用户把鼠标放在组件上时,调用该方法
  this.setState({tmpRating: rating});
}

setRating(rating) { // 用户点击组件时,调用该方法
  this.setState({
    tmpRating: rating,
    rating: rating,
  });
}

reset() { // 用户把鼠标移开时,调用该方法
  this.setTemp(this.state.rating);
}

componentWillReceiveProps(nextProps) { // 响应组件外部的变化
  this.setRating(nextProps.defaultValue);
}
```

最后实现 `render()` 方法，其逻辑包括以下两点。

- 渲染星星的循环，循环次数从 1 开始，到 `this.props.max` 结束。星星可以用符号 `☆` 表示。当星星包含类名 `RatingOn` 时，颜色变为黄色。
- 一个隐藏的 `<input>` 表单域，可以像真正的表单输入框那样使评分的值通过常用的方法取值（和任何普通的 `<input>` 标签一样）：

```
render() {
  const stars = [];
  for (let i = 1; i <= this.props.max; i++) {
    stars.push(
      <span
        className={i <= this.state.tmpRating ? 'RatingOn' : null}
        key={i}
        onClick={!this.props.readonly && this.setRating.bind(this, i)}
        onMouseOver={!this.props.readonly && this.setTemp.bind(this, i)}
      >
        &#9734;
      </span>);
  }
  return (
    <div
      className={classNames({
        'Rating': true,
        'RatingReadonly': this.props.readonly,
      })}
      onMouseOut={this.reset.bind(this)}
    >
```

```
        {stars}
        {this.props.readonly || !this.props.id
          ? null
          : <input
              type="hidden"
              id={this.props.id}
              value={this.state.rating} />
        }
      </div>
    );
  }
```

在这里有一个地方需要注意，那就是 bind 方法的使用。在渲染星星的循环中，绑定当前循环中 i 的值很合理，但为什么要使用 this.reset.bind(this) 呢？事实上，这是在使用 ES class 语法时完成绑定的一种方式。一共有三种方法可以完成绑定：

- 使用 this.method.bind(this)，正如你在上述例子中所看到的那样；
- 使用箭头函数会自动进行绑定，比如 (_unused_event_) => this.method()；
- 在构造函数中一次性绑定。

关于第三种方法，具体做法如下：

```
class Comp extents Component {
  constructor(props) {
   this.method = this.method.bind(this);
  }

  render() {
    return <button onClick={this.method}>
  }
}
```

这种做法的一个优点是，你可以像以前（使用 React.createClass({}) 时）一样，直接使用 this.method 的引用。另一个优点是只需绑定一次即可，不需要在每次调用 render() 方法时都绑定。它的不足之处在于会在控制器中增添更多的模板代码。

### 6.2.9 <FormInput>“工厂组件”

接下来编写一个通用的 <FormInput> 组件，它负责根据传入的属性渲染对应的组件。通过该组件产生的输入组件具有一致的行为（都提供了用于取值的 getValue() 方法）。

在发现工具中编写测试（如图 6-6 所示）：

```
<h2>Form inputs</h2>
<table><tbody>
  <tr>
    <td>Vanilla input</td>
    <td><FormInput /></td>
```

```
    </tr>
    <tr>
      <td>Prefilled</td>
      <td><FormInput defaultValue="it's like a default" /></td>
    </tr>
    <tr>
      <td>Year</td>
      <td><FormInput type="year" /></td>
    </tr>
    <tr>
      <td>Rating</td>
      <td><FormInput type="rating" defaultValue={4} /></td>
    </tr>
    <tr>
      <td>Suggest</td>
      <td><FormInput
        type="suggest"
        options={['red', 'green', 'blue']}
        defaultValue="green" />
      </td>
    </tr>
    <tr>
      <td>Vanilla textarea</td>
      <td><FormInput type="text" /></td>
    </tr>
  </tbody></table>
```

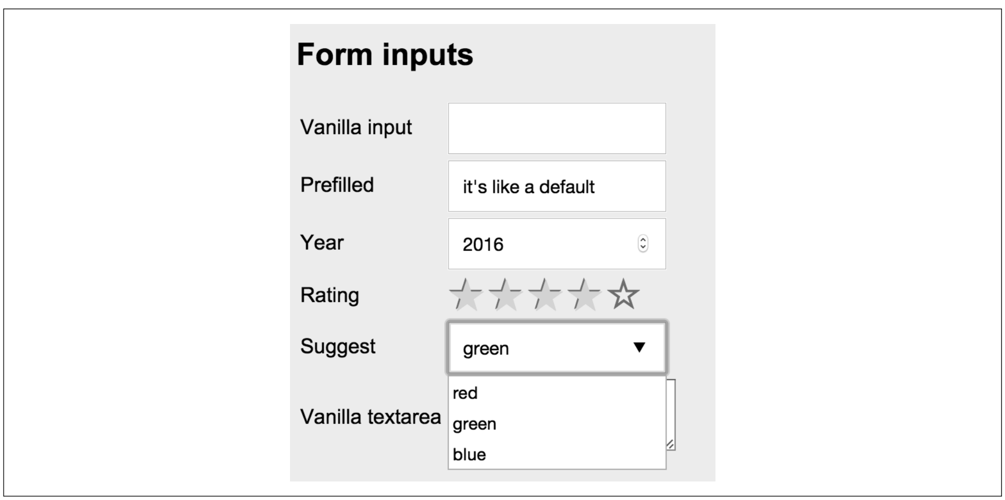



图 6-6：表单输入

`<FormInput>` 组件的实现（js/source/components/FormInput.js）和以往类似，使用了 `import`、`export` 以及用于验证属性的 `propTypes`：

```
import Rating from './Rating';
import React, {Component, PropTypes} from 'react';
```

```
import Suggest from './Suggest';

class FormInput extends Component {
  getValue() {}
  render() {}
}

FormInput.propTypes = {
  type: PropTypes.oneOf(['year', 'suggest', 'rating', 'text', 'input']),
  id: PropTypes.string,
  options: PropTypes.array, // 用于<option>选项列表的自动补全功能
  defaultValue: PropTypes.any,
};

export default FormInput
```

render() 方法包含一个大的 switch 语句块，负责把输入元素创建的工作分配给某一个具体的组件。如果没有匹配到自定义组件，则默认渲染内建的 DOM 元素 <input> 与 <textarea>：

```
render() {
  const common = { // 通用属性
    id: this.props.id,
    ref: 'input',
    defaultValue: this.props.defaultValue,
  };
  switch (this.props.type) {
    case 'year':
      return (
        <input
          {...common}
          type="number"
          defaultValue={this.props.defaultValue || new Date().getFullYear()} />
      );
    case 'suggest':
      return <Suggest {...common} options={this.props.options} />;
    case 'rating':
      return (
        <Rating
          {...common}
          defaultValue={parseInt(this.props.defaultValue, 10)} />
      );
    case 'text':
      return <textarea {...common} />;
    default:
      return <input {...common} type="text" />;
  }
}
```

注意到 ref 属性的使用了吗？事实上，在获取输入框的取值时，ref 属性是比较方便实用的：

```
getValue() {
  return 'value' in this.refs.input
```

```
        ? this.refs.input.value
        : this.refs.input.getValue();
    }
```

在这里，this.refs.input 是底层 DOM 元素的一个引用。对于原生 DOM 元素，比如 <input> 和 <textarea>，通过 this.refs.input.value 获取 value 属性值（就像原生的 DOM 操作方法 document.getElementById('some-input').value 那样）。对于自定义组件，比如 <Suggest> 和 <Rating>，则调用它们本身的 getValue() 方法。

## 6.2.10 <Form>

现在你已经拥有了以下组件：

- 自定义输入元素（比如 <Rating>）；
- 内置的输入元素（比如 <textarea>）；
- <FormInput>，一个根据 type 属性创建输入元素的工厂组件。

是时候建立一个 <Form> 组件把以上组件组合到一起了（如图 6-7 所示）。

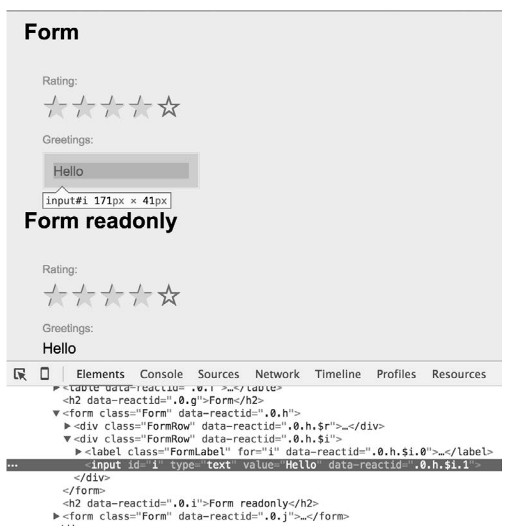



图 6-7：表单

表单组件应该是可重用的，因此不应该把关于这个评价应用的逻辑硬编码到该组件中。（更进一步说，关于酒的一切都不应该被硬编码，这样组件才能重新被应用到供用户抱怨上。）这个 <Form> 组件可以通过 fields 数组进行表单域的配置，而每项表单域的定义内容包括：

- 输入类型 `type`，默认值为 input；
- `id` 属性，以便在随后找到这个输入元素；
- `label` 属性，即输入元素的标签内容；
- 可选属性 `options`，用于传递自动建议的内容列表。

此外，`<Form>` 组件还接收一个包含了默认值的对象，并且配置是否只读的选项，用于阻止用户编辑表单内容：

```
import FormInput from './FormInput';
import Rating from './Rating';
import React, {Component, PropTypes} from 'react';

class Form extends Component {
  getData() {}
  render() {}
}

Form.propTypes = {
  fields: PropTypes.arrayOf(PropTypes.shape({
    id: PropTypes.string.isRequired,
    label: PropTypes.string.isRequired,
    type: PropTypes.string,
    options: PropTypes.arrayOf(PropTypes.string),
  })).isRequired,
  initialData: PropTypes.object,
  readonly: PropTypes.bool,
};

export default Form
```

注意这里使用了 `PropTypes.shape`。该方法具体地表明了对象希望接收的内容。这种用法比起概括性地使用 `fields: PropTypes.arrayOf(PropTypes.object)` 或者 `fields: PropTypes.array` 更为严格，而且当其他开发者开始使用你的组件时，这样做有助于减少错误情况的发生。

`initialData` 是一个键值型的对象字典（`{fieldname: value}`），与组件的 `getData()` 方法所返回的数据格式一致。

以下是 `<Form>` 组件的示例用法，放置在发现工具中：

```
<Form
  fields={[
    {label: 'Rating', type: 'rating', id: 'rateme'},
    {label: 'Greetings', id: 'freetext'},
  ]}
  initialData={ {rateme: 4, freetext: 'Hello'} } />
```

现在回到组件的逻辑实现。这个组件还需要实现 `getData()` 与 `render()` 方法：

```
getData() {
  let data = {};
  this.props.fields.forEach(field =>
    data[field.id] = this.refs[field.id].getValue()
  );
  return data;
}
```

如你所见，这个方法只需要把 render() 方法中设置的 ref 属性循环一遍，并且调用输入元素的 getValue() 方法。

render() 方法本身也相当简单，没有用到任何新语法或新特性：

```
render() {
  return (
    <form className="Form">{this.props.fields.map(field => {
      const prefilled = this.props.initialData && this.props.initial
Data[field.id];
      if (!this.props.readonly) {
        return (
          <div className="FormRow" key={field.id}>
            <label className="FormLabel" htmlFor={field.id}>{field.label}:</
label>
            <FormInput {...field} ref={field.id} defaultValue={prefilled} />
          </div>
        );
      }
      if (!prefilled) {
        return null;
      }
      return (
        <div className="FormRow" key={field.id}>
          <span className="FormLabel">{field.label}:</span>
          {
            field.type === 'rating'
              ? <Rating readonly={true} defaultValue={parseInt(prefilled,
10)} />
              : <div>{prefilled}</div>
          }
        </div>
      );
    }, this)}</form>
  );
}
```

## 6.2.11 <Actions>

接下来需要关注表格中的行。表格的每一行都应该可以进行一些操作（如图 6-8 所示），包括：删除、编辑与查看（当信息没有在行内显示完全时，点击该按钮显示完整内容）。

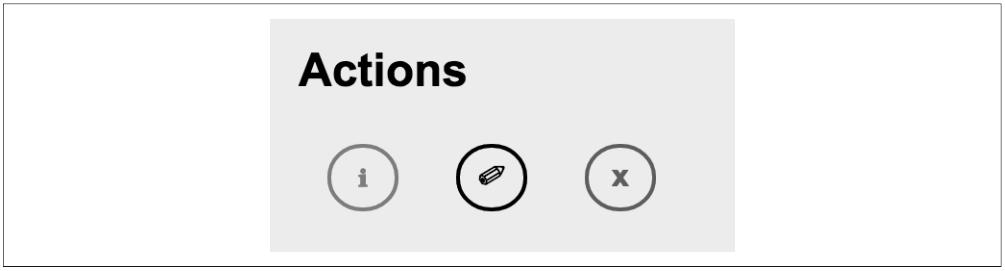



图 6-8：操作

以下是 Actions 组件在发现工具中的测试用例：

```
<h2>Actions</h2>
<div><Actions onAction={type => alert(type)} /></div>
```

其具体实现也相当容易：

```
import React, {PropTypes} from 'react';

const Actions = props =>
  <div className="Actions">
    <span
      tabIndex="0"
      className="ActionsInfo"
      title="More info"
      onClick={props.onAction.bind(null, 'info')}>&#8505;</span>
    <span
      tabIndex="0"
      className="ActionsEdit"
      title="Edit"
      onClick={props.onAction.bind(null, 'edit')}>&#10000;</span>
    <span
      tabIndex="0"
      className="ActionsDelete"
      title="Delete"
      onClick={props.onAction.bind(null, 'delete')}>x</span>
  </div>

Actions.propTypes = {
  onAction: PropTypes.func,
};

Actions.defaultProps = {
  onAction: () => {},
};

export default Actions
```

Actions 是一个简单的组件，只需实现 render() 方法且不需要维护状态。因此可以通过箭头函数将其定义为无状态函数式组件。此外，我们还使用了最简洁的语法：没有 return，

没有 {}，没有 function 语句。（在使用旧语法的日子里，我们大概很难辨认出这是一个函数吧！）

该组件的调用者可以通过 onAction 属性注册回调函数，以监听动作的发生。这种模式也相当简单，作用是让子组件通知父组件有变化发生。如你所见，添加自定义事件（比如 onAction、onAlienAttack 等）就是如此轻松。

## 6.2.12　对话框

接下来建立一个通用的对话框组件，用于显示所有的消息通知（代替 alert()）或弹出窗口（如图 6-9 所示）。比如在添加 / 编辑表单时，需要在数据表格的顶部显示一个模态对话框。

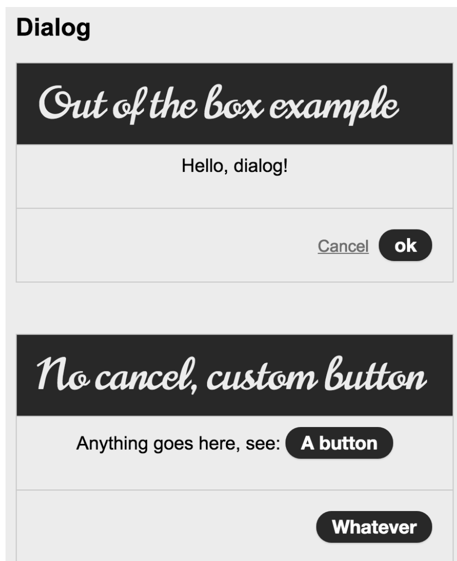



图 6-9：对话框

具体用法：

```
<Dialog
  header="Out-of-the-box example"
  onAction={type => alert(type)}>
    Hello, dialog!
</Dialog>

<Dialog
  header="No cancel, custom button"
  hasCancel={false}
  confirmLabel="Whatever"
  onAction={type => alert(type)}>
```

```
    Anything goes here, see:
    <Button>A button</Button>
</Dialog>
```

该组件的实现方式和 <Actions> 类似——没有状态（只需实现 render() 方法），而且用户点击对话框的底部按钮时，将会触发 onAction 回调函数。

```
import Button from './Button';
import React, {Component, PropTypes} from 'react';

class Dialog extends Component {

}

Dialog.propTypes = {
  header: PropTypes.string.isRequired,
  confirmLabel: PropTypes.string,
  modal: PropTypes.bool,
  onAction: PropTypes.func,
  hasCancel: PropTypes.bool,
};

Dialog.defaultProps = {
  confirmLabel: 'ok',
  modal: false,
  onAction: () => {},
  hasCancel: true,
};

export default Dialog
```

然而，这个组件是通过类定义的，而非箭头函数，因为该组件需要定义两个额外的生命周期方法：

```
componentWillUnmount() {
  document.body.classList.remove('DialogModalOpen');
}

componentDidMount() {
  if (this.props.modal) {
    document.body.classList.add('DialogModalOpen');
  }
}
```

在显示模态对话框时，需要给 document.body 添加一个类名，以便控制整个页面的样式（添加一个灰色的蒙层）。

最后，render() 方法负责把模态框的包裹层、头部、内容区域以及底部组合起来。内容区域可以容纳其他任何组件（或者纯文本）；对话框本身没有对内容作过多限制：

```
render() {
  return (
    <div className={this.props.modal ? 'Dialog DialogModal' : 'Dialog'}>
      <div className={this.props.modal ? 'DialogModalWrap' : null}>
        <div className="DialogHeader">{this.props.header}</div>
        <div className="DialogBody">{this.props.children}</div>
        <div className="DialogFooter">
          {this.props.hasCancel
            ? <span
                className="DialogDismiss"
                onClick={this.props.onAction.bind(this, 'dismiss')}>
                Cancel
              </span>
            : null
          }
          <Button onClick={this.props.onAction.bind(this,
            this.props.hasCancel ? 'confirm' : 'dismiss')}>
            {this.props.confirmLabel}
          </Button>
        </div>
      </div>
    </div>
  );
}
```

此外还有一些优化点，供读者思考。

- 除了提供单个 onAction 属性以外，另一种方案是分别提供 onConfirm（用户点击确认按钮触发）以及 onDismiss（用户点击取消按钮触发）。
- 一个可优化的点是在用户按下 Esc 键时关闭模态框。你有办法实现这个功能吗？
- 包裹层的 div 的类名需要进行条件判断，可以借助 classnames 模块进行优化，用法如下所示。

修改前：

```
<div className={this.props.modal ? 'Dialog DialogModal' : 'Dialog'}>
```

修改后：

```
<div className={classNames({
    'Dialog': true,
    'DialogModal': this.props.modal,
  })}>
```

## 6.3 应用配置

目前为止，所有底层组件都已经完成了。还剩下两个组件需要开发，分别是改进版本的数据表格 Excel 以及顶层组件 Whinepad。这两者都需要借助一个 schema 对象进行配置，它用

于描述你希望在应用中处理的数据类型。针对这个评酒应用，以下是一份 schema 示例代码（js/source/schema.js）：

```
import classification from './classification';

export default [
  {
    id: 'name',
    label: 'Name',
    show: true, // 设置是否在Excel表格中显示
    sample: '$2 chuck',
    align: 'left', // 设置对齐方式
  },
  {
    id: 'year',
    label: 'Year',
    type: 'year',
    show: true,
    sample: 2015,
  },
  {
    id: 'grape',
    label: 'Grape',
    type: 'suggest',
    options: classification.grapes,
    show: true,
    sample: 'Merlot',
    align: 'left',
  },
  {
    id: 'rating',
    label: 'Rating',
    type: 'rating',
    show: true,
    sample: 3,
  },
  {
    id: 'comments',
    label: 'Comments',
    type: 'text',
    sample: 'Nice for the price',
  },
]
```

这个示例的用法是 ECMAScript 模块中你可以想到的最简单的一种形式——只输出一个变量。这个模块还引入了另一个简单的模块，里面包含一长串用于预填充表单的选项（js/source/classification.js）：

```
export default {
  grapes: [
    'Baco Noir',
    'Barbera',
```

```
      'Cabernet Franc',
      'Cabernet Sauvignon',
      // ....
    ],
  }
```

借助 schema 模块，现在你就可以配置应用中所需的数据类型了。

## 6.4　<Excel>：改进的新版本

我们在第 3 章中创建的 Excel 组件承载了太多功能。因此我们需要创建一个改进的新版本，以提高组件的可重用性。我们决定删减搜索功能（把功能移到顶层的 <Whinepad> 中）与下载功能（如果你喜欢的话也可以保留下来）。这个组件的全部功能应该与 CRUD 中的 RUD 部分相关（如图 6-10 所示）。由于这是一个可编辑的表格，我们还需要新增 onDataChange 属性，以便在表格中的数据内容发生改变时通知父组件 Whinepad。




图 6-10：Excel 组件

Whinepad 组件则负责搜索功能，CRUD 中的 C 部分（创建新条目）以及使用 localStorage 进行数据持久化存储。（在实际应用开发中，你也可能需要把数据存储在服务器上。）

这两个组件都需要使用 schema 对象进行数据类型配置。

以下是 Excel 组件的完整实现（和第 3 章的版本类似，其中某些功能稍有不同）：

```
import Actions from './Actions';
import Dialog from './Dialog';
import Form from './Form';
import FormInput from './FormInput';
import Rating from './Rating';
import React, {Component, PropTypes} from 'react';
import classNames from 'classnames';

class Excel extends Component {

  constructor(props) {
    super(props);
    this.state = {
      data: this.props.initialData,
      sortby: null, // schema.id
```

```
      descending: false,
      edit: null, // [row index, schema.id],
      dialog: null, // {type, idx}
    };
  }

  componentWillReceiveProps(nextProps) {
    this.setState({data: nextProps.initialData});
  }

  _fireDataChange(data) {
    this.props.onDataChange(data);
  }

  _sort(key) {
    let data = Array.from(this.state.data);
    const descending = this.state.sortby === key && !this.state.descending;
    data.sort(function(a, b) {
      return descending
        ? (a[column] < b[column] ? 1 : -1)
        : (a[column] > b[column] ? 1 : -1);
    });
    this.setState({
      data: data,
      sortby: key,
      descending: descending,
    });
    this._fireDataChange(data);
  }

  _showEditor(e) {
    this.setState({edit: {
      row: parseInt(e.target.dataset.row, 10),
      key: e.target.dataset.key,
    }});
  }

  _save(e) {
    e.preventDefault();
    const value = this.refs.input.getValue();
    let data = Array.from(this.state.data);
    data[this.state.edit.row][this.state.edit.key] = value;
    this.setState({
      edit: null,
      data: data,
    });
    this._fireDataChange(data);
  }

  _actionClick(rowidx, action) {
    this.setState({dialog: {type: action, idx: rowidx}});
  }

  _deleteConfirmationClick(action) {
    if (action === 'dismiss') {
```

```
    this._closeDialog();
    return;
  }
  let data = Array.from(this.state.data);
  data.splice(this.state.dialog.idx, 1);
  this.setState({
    dialog: null,
    data: data,
  });
  this._fireDataChange(data);
}

_closeDialog() {
  this.setState({dialog: null});
}

_saveDataDialog(action) {
  if (action === 'dismiss') {
    this._closeDialog();
    return;
  }
  let data = Array.from(this.state.data);
  data[this.state.dialog.idx] = this.refs.form.getData();
  this.setState({
    dialog: null,
    data: data,
  });
  this._fireDataChange(data);
}

render() {
  return (
    <div className="Excel">
      {this._renderTable()}
      {this._renderDialog()}
    </div>
  );
}

_renderDialog() {
  if (!this.state.dialog) {
    return null;
  }
  switch (this.state.dialog.type) {
    case 'delete':
      return this._renderDeleteDialog();
    case 'info':
      return this._renderFormDialog(true);
    case 'edit':
      return this._renderFormDialog();
    default:
      throw Error(`Unexpected dialog type ${this.state.dialog.type}`);
  }
}
```

```
  _renderDeleteDialog() {
    const first = this.state.data[this.state.dialog.idx];
    const nameguess = first[Object.keys(first)[0]];
    return (
      <Dialog
        modal={true}
        header="Confirm deletion"
        confirmLabel="Delete"
        onAction={this._deleteConfirmationClick.bind(this)}
      >
        {`Are you sure you want to delete "${nameguess}"?`}
      </Dialog>
    );
  }

  _renderFormDialog(readonly) {
    return (
      <Dialog
        modal={true}
        header={readonly ? 'Item info' : 'Edit item'}
        confirmLabel={readonly ? 'ok' : 'Save'}
        hasCancel={!readonly}
        onAction={this._saveDataDialog.bind(this)}
      >
        <Form
          ref="form"
          fields={this.props.schema}
          initialData={this.state.data[this.state.dialog.idx]}
          readonly={readonly} />
      </Dialog>
    );
  }

  _renderTable() {
    return (
      <table>
        <thead>
          <tr>{
            this.props.schema.map(item => {
              if (!item.show) {
                return null;
              }
              let title = item.label;
              if (this.state.sortby === item.id) {
                title += this.state.descending ? ' \u2191' : ' \u2193';
              }
              return (
                <th
                  className={`schema-${item.id}`}
                  key={item.id}
                  onClick={this._sort.bind(this, item.id)}
                >
                  {title}
                </th>
              );
```

```
          }, this)
        }
        <th className="ExcelNotSortable">Actions</th>
        </tr>
      </thead>
      <tbody onDoubleClick={this._showEditor.bind(this)}>
        {this.state.data.map((row, rowidx) => {
          return (
            <tr key={rowidx}>{
              Object.keys(row).map((cell, idx) => {
                const schema = this.props.schema[idx];
                if (!schema || !schema.show) {
                  return null;
                }
                const isRating = schema.type === 'rating';
                const edit = this.state.edit;
                let content = row[cell];
                if (!isRating && edit && edit.row === rowidx && edit.key ===
schema.id) {
                  content = (
                    <form onSubmit={this._save.bind(this)}>
                      <FormInput ref="input" {...schema} defaultValue={con
tent} />
                    </form>
                  );
                } else if (isRating) {
                  content = <Rating readonly={true} defaultValue={Num
ber(content)} />;
                }
                return (
                  <td
                    className={classNames({
                      [`schema-${schema.id}`]: true,
                      'ExcelEditable': !isRating,
                      'ExcelDataLeft': schema.align === 'left',
                      'ExcelDataRight': schema.align === 'right',
                      'ExcelDataCenter': schema.align !== 'left' &&
schema.align !== 'right',
                    })}
                    key={idx}
                    data-row={rowidx}
                    data-key={schema.id}>
                    {content}
                  </td>
                );
              }, this)}
              <td className="ExcelDataCenter">
                <Actions onAction={this._actionClick.bind(this, rowidx)} />
              </td>
            </tr>
          );
        }, this)}
      </tbody>
    </table>
  );
```

```
  }
}

Excel.propTypes = {
  schema: PropTypes.arrayOf(
    PropTypes.object
  ),
  initialData: PropTypes.arrayOf(
    PropTypes.object
  ),
  onDataChange: PropTypes.func,
};

export default Excel
```

有一些细节需要详细讨论：

```
render() {
  return (
    <div className="Excel">
      {this._renderTable()}
      {this._renderDialog()}
    </div>
  );
}
```

这个组件会渲染一个表格与一个对话框（是否渲染对话框视情况而定）。对话框的情况多种多样，包括弹出确认信息（sure you want to delete?）、编辑表单、在表单只读时显示条目信息。对话框在默认情况下不会显示，可以通过设置 `this.state` 的 `dialog` 属性在需要时渲染对话框，状态变化将会导致组件重新渲染。

当用户点击 `<Action>` 组件中的某个按钮时，你需要设置 `dialog` 属性：

```
_actionClick(rowidx, action) {
  this.setState({dialog: {type: action, idx: rowidx}});
}
```

当表格中的数据发生变化时（即使用 `this.setState({data: /**/})` 的时候），你需要触发一个监听改变的事件，通知父组件更新持久化存储中的内容：

```
_fireDataChange(data) {
  this.props.onDataChange(data);
}
```

反向通信（从父组件 `Whinepad` 到子组件 `Excel` 之间的通信）会在父组件改变 `initialData` 属性时发生。`Excel` 组件可以通过以下方法响应数据变化：

```
componentWillReceiveProps(nextProps) {
  this.setState({data: nextProps.initialData});
}
```

如何创建数据条目（如图 6-11 所示）或一个数据视图（如图 6-12 所示）？你需要打开一个包含 Form 组件的 Dialog 对话框。表单中的数据配置来源于 schema，而数据条目的内容则来源于 this.state.data：

```
_renderFormDialog(readonly) {
  return (
    <Dialog
      modal={true}
      header={readonly ? 'Item info' : 'Edit item'}
      confirmLabel={readonly ? 'ok' : 'Save'}
      hasCancel={!readonly}
      onAction={this._saveDataDialog.bind(this)}
    >
      <Form
        ref="form"
        fields={this.props.schema}
        initialData={this.state.data[this.state.dialog.idx]}
        readonly={readonly} />
    </Dialog>
  );
}
```

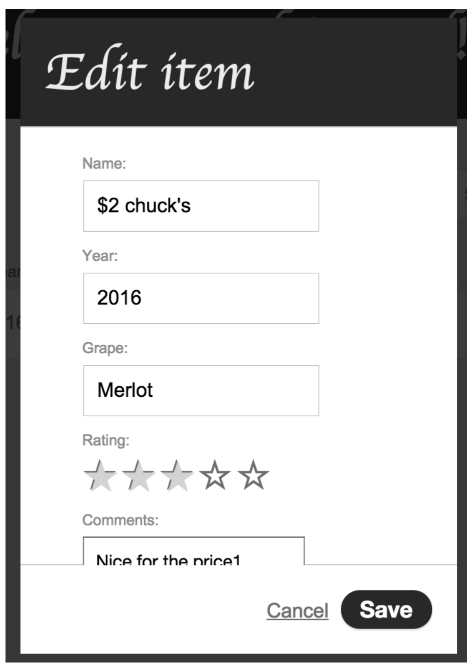



图 6-11：编辑数据的对话框（CRUD 中的 U）

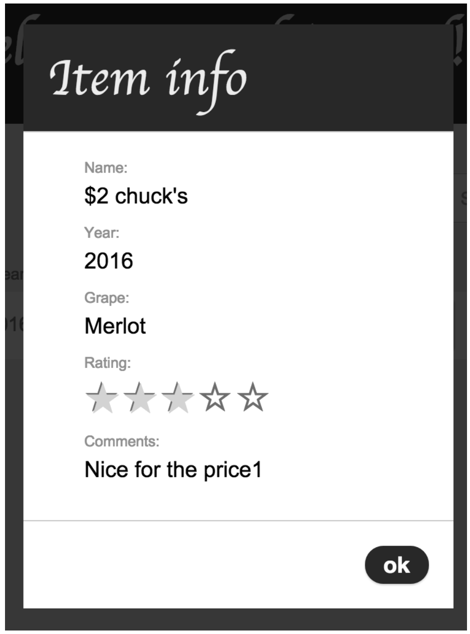



图 6-12：数据视图对话框（CRUD 中的 R）

当用户完成编辑后，你需要更新状态，并把变化告知订阅者：

```
_saveDataDialog(action) {
  if (action === 'dismiss') {
    this._closeDialog(); // 只需要把this.state.dialog设置为空
    return;
  }
  let data = Array.from(this.state.data);
  data[this.state.dialog.idx] = this.refs.form.getData();
  this.setState({
    dialog: null,
    data: data,
  });
  this._fireDataChange(data);
}
```

至于 ES 的新语法，这里除了模板字符串的广泛使用，并没有涉及太多：

```
// 旧语法
"Are you sure you want to delete " + nameguess + "?"

// 新语法
{`Are you sure you want to delete "${nameguess}"?`}
```

此外还要注意类名中也使用了模板字符串，因为这个应用允许你在 `schema` 中通过添加 ID 的方式自定义数据表格：

```
// 旧语法
<th className={"schema-" + item.id}}>

// 新语法
<th className={`schema-${item.id}`}>
```

模板字符串最令人不可思议的用法，是可以通过中括号 [] 用作对象中的属性名。虽然这与 React 本身没有关系，但当你看到如下用法时，可能还是会感到有些奇怪：

```
{
  [`schema-${schema.id}`]: true,
  'ExcelEditable': !isRating,
  'ExcelDataLeft': schema.align === 'left',
  'ExcelDataRight': schema.align === 'right',
  'ExcelDataCenter': schema.align !== 'left' && schema.align !== 'right',
}
```

## 6.5 <Whinepad>

终于到了最后一个组件，也就是所有组件的父组件（如图 6-13 所示）。这个组件比 Excel 表格组件简单一点，依赖也更少：

```
import Button from './Button'; // <- 用于添加新条目
import Dialog from './Dialog'; // <- 用于弹出添加新条目的对话框
import Excel from './Excel'; // <- 所有内容的表格容器
import Form from './Form'; // <- 添加新条目时的表单
import React, {Component, PropTypes} from 'react';
```

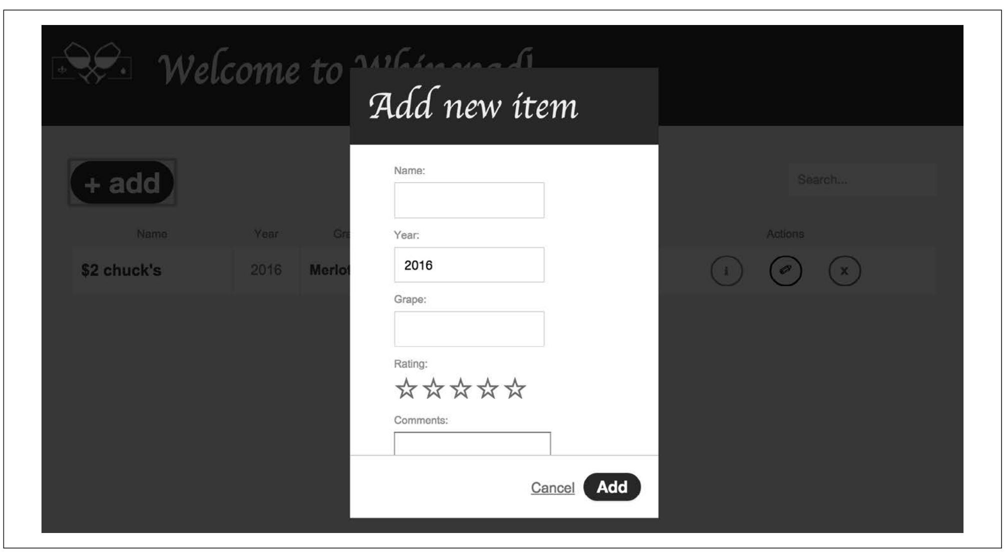



图 6-13：Whinepad 负责 CRUD 中的 C

这个组件只需接收两个属性：schema 数据配置和初始数据：

```
Whinepad.propTypes = {
  schema: PropTypes.arrayOf(
    PropTypes.object
  ),
  initialData: PropTypes.arrayOf(
    PropTypes.object
  ),
};

export default Whinepad;
```

如果你已经仔细阅读了 Excel 组件的实现逻辑，这个组件对你来说应该不太困难：

```
class Whinepad extends Component {

  constructor(props) {
    super(props);
    this.state = {
      data: props.initialData,
      addnew: false,
    };
    this._preSearchData = null;
  }

  _addNewDialog() {
    this.setState({addnew: true});
  }

  _addNew(action) {
    if (action === 'dismiss') {
      this.setState({addnew: false});
      return;
    }
    let data = Array.from(this.state.data);
    data.unshift(this.refs.form.getData());
    this.setState({
      addnew: false,
      data: data,
    });
    this._commitToStorage(data);
  }

  _onExcelDataChange(data) {
    this.setState({data: data});
    this._commitToStorage(data);
  }

  _commitToStorage(data) {
    localStorage.setItem('data', JSON.stringify(data));
  }

  _startSearching() {
```

```
    this._preSearchData = this.state.data;
  }

  _doneSearching() {
    this.setState({
      data: this._preSearchData,
    });
  }

  _search(e) {
    const needle = e.target.value.toLowerCase();
    if (!needle) {
      this.setState({data: this._preSearchData});
      return;
    }
    const fields = this.props.schema.map(item => item.id);
    const searchdata = this._preSearchData.filter(row => {
      for (let f = 0; f < fields.length; f++) {
        if (row[fields[f]].toString().toLowerCase().indexOf(needle) > -1) {
          return true;
        }
      }
      return false;
    });
    this.setState({data: searchdata});
  }

  render() {
    return (
      <div className="Whinepad">
        <div className="WhinepadToolbar">
          <div className="WhinepadToolbarAdd">
            <Button
              onClick={this._addNewDialog.bind(this)}
              className="WhinepadToolbarAddButton">
              + add
            </Button>
          </div>
          <div className="WhinepadToolbarSearch">
            <input
              placeholder="Search..."
              onChange={this._search.bind(this)}
              onFocus={this._startSearching.bind(this)}
              onBlur={this._doneSearching.bind(this)} />
          </div>
        </div>
        <div className="WhinepadDatagrid">
          <Excel
            schema={this.props.schema}
            initialData={this.state.data}
            onDataChange={this._onExcelDataChange.bind(this)} />
        </div>
        {this.state.addnew
          ? <Dialog
              modal={true}
```

```
              header="Add new item"
              confirmLabel="Add"
              onAction={this._addNew.bind(this)}
            >
              <Form
                ref="form"
                fields={this.props.schema} />
            </Dialog>
          : null}
      </div>
    );
  }
}
```

要注意组件通过 onDataChange 注册来监听 Excel 组件中的数据变化。还要知道所有数据都只是简单地存储在 localStorage 中：

```
_commitToStorage(data) {
  localStorage.setItem('data', JSON.stringify(data));
}
```

此处也可以发起任何异步请求（即 XHR、XMLHttpRequest、Ajax），把数据保存在服务端，而不仅是保存在客户端。

# 6.6　总结

本章开始就提到，应用的主入口文件是 app.js。这个文件既不是组件也不是模块；它没有输出任何内容。它的作用仅仅是进行初始化工作——从 localStorage 中读取已存在的数据，并配置 <Whinepad> 组件：

```
'use strict';

import Logo from './components/Logo';
import React from 'react';
import ReactDOM from 'react-dom';
import Whinepad from './components/Whinepad';
import schema from './schema';

let data = JSON.parse(localStorage.getItem('data'));

// 如果localStorage中没有数据,则从schema中读取默认的示例数据
if (!data) {
  data = {};
  schema.forEach(item => data[item.id] = item.sample);
  data = [data];
}

ReactDOM.render(
  <div>
    <div className="app-header">
```

```
      <Logo /> Welcome to Whinepad!
    </div>
    <Whinepad schema={schema} initialData={data} />
  </div>,
  document.getElementById('pad')
);
```

至此，这个应用就完成了。你可以到 http://whinepad.com 进行体验，并在 https://github.com/stoyan/reactbook/ 浏览完整代码。

第 7 章

# lint、Flow、测试与复验

随后的第 8 章将会介绍 Flux，它是管理组件间通信的另一种选择（用于代替 `onDataChange` 这样的方法）。届时我们需要对代码进行一点重构。如果在重构时能尽可能地避免错误的发生，不是更好吗？在本章中，我们将介绍几个工具，帮助你在应用规模不可避免地增长时保持头脑清醒。这些工具就是 ESLint、Flow 和 Jest。

不过，使用它们的共同前提是配置 package.json 文件。

## 7.1 package.json

前面提到过使用 `npm` 安装第三方库与工具的方法。除此之外，`npm` 还允许你把项目打包并共享到 http://npmjs.com，让其他用户可以通过 `npm` 安装你的软件包。然而，你不一定非要把代码上传到 npmjs.com 才能利用 `npm` 提供的一些便利之处。

打包工作是围绕 package.json 文件进行的，你可以把该文件放在项目根目录，并在该文件中配置依赖与其他附加工具。该文件拥有非常丰富的可配置选项（参见 https://docs.npmjs.com/files/package.json 获取完整内容），但我们首先需要关心如何使用这个文件，并编写最小限度的必要配置。

在应用目录中新建一个文件，命名为 package.json：

```
$ cd ~/reactbook/whinepad2
$ touch package.json
```

在文件中添加以下内容：

```
{
  "name": "whinepad",
  "version": "2.0.0",
}
```

准备工作完成了。接下来，只需要往该文件中不断添加更多配置项即可。

### 7.1.1 配置 Babel

在第 5 章中，build.sh 通过以下命令运行 Babel：

```
$ babel --presets react,es2015 js/source -d js/build
```

你可以把命令中的预设配置转移到 package.json 文件中，从而简化该命令：

```
{
  "name": "whinepad",
  "version": "2.0.0",
  "babel": {
      "presets": [
          "es2015",
          "react"
      ]
  },
}
```

现在只需要输入以下命令即可运行 Babel：

```
$ babel js/source -d js/build
```

Babel（以及 JavaScript 生态圈中的很多工具）会检查 package.json 文件是否存在；如果存在，就从该文件中读取配置选项。

### 7.1.2 脚本

NPM 允许你在 package.json 中配置脚本，并通过 `npm run scriptname` 的方式运行指定脚本。举个例子，我们把第 5 章 ./scripts/watch.sh 中的一行命令移到 package.json 中：

```
{
  "name": "whinepad",
  "version": "2.0.0",
  "babel": {/* ... */},
  "scripts": {
    "watch": "watch \"sh scripts/build.sh\" js/source css/"
  }
}
```

要运行这个构建脚本，可以通过：

```
# 过去
$ sh ./scripts/watch.sh

# 现在
$ npm run watch
```

如果你想继续改进这些脚本的话，可以通过相同的方式，把 build.sh 的内容移动到 package.json 中。你也可以选择使用某个构建工具（比如 Grunt、Gulp 等），它们也可以通过 package.json 进行配置。但本书只讨论与 React 相关的话题，此处不展开讨论。目前我们已经总结了关于 package.json 文件需要掌握的所有内容。

## 7.2 ESLint

ESLint（http://eslint.org/）是一个 JavaScript 和 JSX 检查工具，可以帮助你检查代码中的潜在问题。此外，ESLint 还可以帮助你检查代码一致性，比如缩进与空格的使用。这个工具还可以检查拼写错误以及未使用的变量。在理想情况下，ESLint 除了作为构建过程中的一部分，还应该被整合到你的源代码控制系统以及代码编辑器中，以保证 ESLint 最大限度地提醒你少犯错误。

### 7.2.1 安装

除了 ESLint 本身，你还需要安装 React 和 Babel 的插件，用于帮助 ESLint 识别 ECMAScript 的最新语法，以及从 JSX 与 React 的特定“规则”中受益：

```
$ npm i -g eslint babel-eslint eslint-plugin-react eslint-plugin-babel
```

然后在 package.json 文件中添加 `eslintConfig` 配置：

```
{
  "name": "whinepad",
  "version": "2.0.0",
  "babel": {},
  "scripts": {},
  "eslintConfig": {
    "parser": "babel-eslint",
    "plugins": [
      "babel",
      "react"
    ],
  }
}
```

### 7.2.2 运行

对单个文件运行检查工具的命令如下：

```
$ eslint js/source/app.js
```

理想状况下，这条命令应该不会提示错误信息。这意味着 ESLint 可以正常识别 JSX 以及其他古怪的语法。但是这还不够好，因为目前检查工具还没有验证任何规则。ESLint 会根据预先定义的规则逐一检查每项内容。在起步阶段，你可以（通过 extend）使用 ESLint 内置的一些规则：

```
"eslintConfig": {
  "parser": "babel-eslint",
  "plugins": [],
  "extends": "eslint:recommended"
}
```

再次运行该命令，你会得到一些错误提示信息：

```
$ eslint js/source/app.js

/Users/stoyanstefanov/reactbook/whinepad2/js/source/app.js
   4:8   error  "React" is defined but never used  no-unused-vars
   9:23  error  "localStorage" is not defined      no-undef
  25:3   error  "document" is not defined          no-undef

✖ 3 problems (3 errors, 0 warnings)
```

第二条和第三条错误信息都是关于未声明的变量（来源于规则 `no-undef`），但实际上这些变量都是可以在浏览器中全局访问的。因此可以通过在 `eslintConfig` 中增加配置项，以修复该问题：

```
"env": {
  "browser": true
}
```

第一条错误信息和 React 相关。虽然你的确需要引入 `React`，但从 ESLint 的角度看来，这似乎是一个没有使用过的变量，因此没有必要存在。借助 `eslint-plugin-react` 中的一条规则可以解决这个问题：

```
"rules": {
  "react/jsx-uses-react": 1
}
```

接下来检查 schema.js 文件，你会得到另一种类型的错误信息：

```
$ eslint js/source/schema.js

/Users/stoyanstefanov/reactbook/whinepad2/js/source/schema.js
   9:18 error Unexpected trailing comma comma-dangle
  16:17 error Unexpected trailing comma comma-dangle
  25:18 error Unexpected trailing comma comma-dangle
  32:14 error Unexpected trailing comma comma-dangle
```

```
  38:33 error Unexpected trailing comma comma-dangle
  39:4  error Unexpected trailing comma comma-dangle

✖ 6 problems (6 errors, 0 warnings)
```

末尾逗号（比如 `let a = [1,]`，而非 `let a = [1]`）有时是不好的（因为在某些老版本的浏览器中，末尾逗号会导致语法错误），但是保留末尾逗号也能带来一些便利，因为其有助于源代码控制系统追溯（blame）提交历史，并且方便修改文件。通过对配置文件进行一点改动，可以让（在数组或对象占据了多行的情况下）总是使用末尾逗号变成一种好的做法：

```
"rules": {
  "comma-dangle": [2, "always-multiline"],
  "react/jsx-uses-react": 1
}
```

### 7.2.3　规则列表

要获取完整的规则列表，可以在本书附带的代码库（https://github.com/stoyan/reactbook/）中找到。这份规则（本书项目遵循的规则）也是 React 库的源代码本身遵循的规则列表。

最后把语法检查命令添加到 build.sh 脚本中。这样做可以让 ESLint 在构建时帮助你检查代码，以保证维持代码的高质量：

```
# QA
eslint js/source
```

## 7.3　Flow

Flow（http://flowtype.org）是针对 JavaScript 的静态类型检查工具。总体而言，人们对于类型有两种分化的意见，特别是在 JavaScript 领域。

一些人喜欢自己的代码得到监控，确保程序处理的是正确的数据。就像语法检查和单元测试那样，自动检测代码可以确保正确处理数据，在一定程度上保证你不会遗漏一些未检查（或者自认为没有问题）的代码。随着应用规模和开发人员的不断增长，类型检查显得愈发重要。

另一些人则喜欢 JavaScript 这种动态、弱类型的语言，认为类型检测会带来更多麻烦，因为有时候需要进行类型转换。

当然，是否需要安装这个工具完全取决于你和你的团队。如果你有兴趣，不妨尝试探索一下。

## 7.3.1 安装

```
$ npm install -g flow-bin
$ cd ~/reactbook/whinepad2
$ flow init
```

`init` 命令会在你的目录中创建一个空的 .flowconfig 配置文件。在该文件中的 `ignore` 和 `include` 部分添加如下内容：

```
[ignore]
.*/react/node_modules/.*

[include]
node_modules/react
node_modules/react-dom
node_modules/classnames

[libs]
[options]
```

## 7.3.2 运行

要运行 Flow，只需要输入：

```
$ flow
```

要检查单个文件或目录，可以输入：

```
$ flow js/source/app.js
```

最后，把命令添加到构建脚本中，作为质量保证（quality assurance，QA）过程的一部分：

```
# QA
eslint js/source
flow
```

## 7.3.3 注册类型检查

要让 Flow 对你的文件进行类型检查，需要在文件顶部的第一个注释中添加 `@flow` 标记。如果没有在文件中添加该标记，Flow 就会忽略这个文件。也就是说，类型检查完全是可选的。

我们从上一章中最简单的一个组件开始检查，即 `<Button>` 组件：

```
/* @flow */

import classNames from 'classnames';
import React, {PropTypes} from 'react';
```

```
const Button = props =>
  props.href
    ? <a {...props} className={classNames('Button', props.className)} />
    : <button {...props} className={classNames('Button', props.className)} />


Button.propTypes = {
  href: PropTypes.string,
};

export default Button
```

运行 Flow：

```
$ flow js/source/components/Button.js
js/source/components/Button.js:6
  6: const Button = props =>
                    ^^^^^ parameter `props`. Missing annotation

Found 1 error
```

出现了一个错误，但这是好事——至少我们有机会让代码变得更好了！ Flow 的检查结果显示 `props` 参数缺少注解。

比如对于下面这个函数：

```
function sum(a, b) {
  return a + b;
}
```

Flow 希望你为其添加注解：

```
function sum(a: number, b: number): number {
  return a + b;
}
```

从而避免最终不能得到预期的结果：

```
sum('1' + 2); // "12"
```

### 7.3.4　修复 <Button>

我们的函数中接收的 `props` 参数应该是一个对象。因此你可以这样修改：

```
const Button = (props: Object) =>
```

现在 Flow 就不会报错了：

```
$ flow js/source/components
No errors!
```

虽然注解为对象类型 `Object` 是有效的，但你可以做得更加具体，创建一个自定义类型，并指明对象中包含的属性与类型：

```
type Props = {
  href: ?string,
};

const Button = (props: Props) =>
  props.href
    ? <a {...props} className={classNames('Button', props.className)} />
    : <button {...props} className={classNames('Button', props.className)} />

export default Button
```

如你所见，切换到自定义类型后，Flow 就取代了 React 中 `propTypes` 定义的作用。这意味着：

- 不需要在运行时进行类型检查，有助于加快运行速度；
- 可以减少发送到客户端的代码量（减少字节数）。

此外，把属性类型放回组件顶部可以方便我们直接把这段定义看作组件的文档说明。

在 `href: ?string` 中的问号表示这个属性可以为空。

> 既然我们不需要再使用 `propTypes`，ESLint 会提醒我们变量 `PropTypes` 未被使用。因此需要将 `import React, {PropTypes} from 'react';` 改为 `import React from 'react';`。
>
> ESLint 会一直监控代码，并帮助我们改进这些细节上的小疏漏。现在是否感觉到开发体验更棒了？

再次运行 Flow，你会得到另一个错误：

```
$ flow js/source/components/Button.js
js/source/components/Button.js:12
 12:     ? <a {...props} className={classNames('Button', props.className)} />
                                                          ^^^^^^^^^
property `className`.
                                                          Property not
found in
12:      ? <a {...props} className={classNames('Button', props.className)} />
                                                   ^^^^^ object type
```

这里的问题是 Flow 不知道在 `props` 对象中存在 `className` 属性，因为我们在自定义类型 `Props` 中没有指明该属性。要解决这个问题，需要把 `className` 添加到类型中：

```
type Props = {
```

```
    href: ?string,
    className: ?string,
  };
```

### 7.3.5 app.js

对主文件 app.js 运行检查，会得到如下错误：

```
$ flow js/source/app.js
js/source/app.js:11
 11: let data = JSON.parse(localStorage.getItem('data'));
                           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^ call of method `getItem`
 11: let data = JSON.parse(localStorage.getItem('data'));
                           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^ null. This type is
incompatible with
383:     static parse(text: string, reviver?: (key: any, value: any) => any):
any;
                            ^^^^^^ string. See lib: /private/tmp/flow/flow
lib_28f8ac7e/core.js:383
```

Flow 要求你只能传递字符串到 `JSON.parse()` 函数中，并且帮你指出了 `parse()` 函数所在的位置。由于 `localStorage.getItem` 可能会返回 `null`，这是不可接受的。一种简单的解决方案是添加默认值：

```
let data = JSON.parse(localStorage.getItem('data') || '');
```

然而，`JSON.parse('')` 会在浏览器中导致错误（尽管这种写法可以通过类型检查），因为空字符串不是合法的 JSON 编码字符串。因此，需要修改这段代码才能在满足 Flow 进行类型检测的同时避免浏览器报错。

你可能会发现处理类型问题还挺烦人的，但好处是 Flow 会让你好好思考值传递中的问题。

在 app.js 文件中的相关代码是：

```
let data = JSON.parse(localStorage.getItem('data'));

// 从schema中读取默认示例数据
if (!data) {
  data = {};
  schema.forEach((item) => data[item.id] = item.sample);
  data = [data];
}
```

这段代码中的另一个问题是，`data` 一开始是数组，然后变成了对象，最后又变回数组。虽然 JavaScript 允许你这样做，但这似乎是一种反模式——中途改变变量类型。实际上，浏览器内部的 JavaScript 引擎会设置变量类型，目的是优化代码。因此当你中途改变变量类型时，浏览器的优化就不起作用了，这样做可不好。

接下来我们一起修复这些问题。

可以更严格地限制 data 变量，并将其注解为一个对象数组：

```
let data: Array<Object>;
```

然后把从数据存储中读取出来的变量称为 storage，其类型为字符串（也可能为 null，因此需要在前面加上问号）：

```
const storage: ?string = localStorage.getItem('data');
```

当 storage 为字符串时，只需要直接解析就可以了。否则，你需要设置 data 为数组类型，并把示例数据填充到数组的第一个元素中：

```
if (!storage) {
  data = [{}];
  schema.forEach(item => data[0][item.id] = item.sample);
} else {
  data = JSON.parse(storage);
}
```

现在这两个文件都已经兼容 Flow 了。为了节省篇幅，不再列出所有经过兼容处理的代码；接下来我们将关注 Flow 的几个更加有趣的特性。本书附带的代码库（https://github.com/stoyan/reactbook/）中包含了完整代码。

### 7.3.6　关于 props 和 state 类型检查的更多内容

当你使用无状态函数创建 React 组件时，可以像之前看到的那样，为 props 参数加上注解：

```
type Props = {/* ... */};
const Button = (props: Props) => {/* ... */};
```

类构造函数与之类似：

```
type Props = {/* ... */};
class Rating extends Component {
  constructor(props: Props) {/* ... */}
}
```

但如果不使用构造函数呢？比如这样：

```
class Form extends Component {
  getData(): Object {}
  render() {}
}
```

在这种情况下，可以使用 ECMAScript 的类属性功能：

```
type Props = {/* ... */};
class Form extends Component {
  props: Props;
  getData(): Object {}
  render() {}
}
```

在本书编写时，类属性还没有被正式纳入 ECMAScript 标准，不过好在 Babel 提供了支持最新特性的 `stage-0` 预设。在使用前，你需要安装 `babel-preset-stage-0` 这个 NPM 包，并更新 package.json 文件中的 Babel 配置：

```
{
  "babel": {
    "presets": [
      "es2015",
      "react",
      "stage-0"
    ]
  }
}
```

类似地，你可以注解组件中的 state，并把类型定义放在组件代码的顶部。除了有助于类型检查，state 定义前置还可以作为组件的文档使用，帮助你定位组件的 bug。例如：

```
type Props = {
  defaultValue: number,
  readonly: boolean,
  max: number,
};

type State = {
  rating: number,
  tmpRating: number,
};

class Rating extends Component {
  props: Props;
  state: State;
  constructor(props: Props) {
    super(props);
    this.state = {
      rating: props.defaultValue,
      tmpRating: props.defaultValue,
    };
  }
}
```

当然，你也应该尽可能地利用这个自定义的类型：

```
componentWillReceiveProps(nextProps: Props) {
  this.setRating(nextProps.defaultValue);
}
```

### 7.3.7　导出 / 导入类型

接下来看一下 <FormInput> 组件：

```
type FormInputFieldType = 'year' | 'suggest' | 'rating' | 'text' | 'input';

export type FormInputFieldValue = string | number;

export type FormInputField = {
  type: FormInputFieldType,
  defaultValue?: FormInputFieldValue,
  id?: string,
  options?: Array<string>,
  label?: string,
};

class FormInput extends Component {
  props: FormInputField;
  getValue(): FormInputFieldValue {}
  render() {}
}
```

在这段代码中，你可以看到注解是如何配置可选值的，其作用类似于 React 中的 `oneOf()` 属性类型。

你还可以看到如何把一种自定义类型（`FormInputFieldType`）作为另一种自定义类型（`FormInputField`）的一部分。

最后需要导出（export）这些类型。这样做的话，当另一个组件使用同一种类型时，就不需要再重新定义了，只需要导入（import）该组件导出的类型即可。下面的例子展示了 <Form> 组件如何使用 <FormInput> 提供的类型：

```
import type FormInputField from './FormInput';

type Props = {
  fields: Array<FormInputField>,
  initialData?: Object,
  readonly?: boolean,
};
```

实际上，表单需要同时使用 `FormInput` 中的两种类型，因此可以这样写：

```
import type {FormInputField, FormInputFieldValue} from './FormInput';
```

### 7.3.8 类型转换

Flow 允许你为值指定一个不同于 Flow 所推导出的类型。举个例子，你在事件处理函数中接收一个事件对象，而 Flow 推导出事件的 `target` 属性类型和你的预想有出入。思考 `Excel` 组件中的这段代码：

```
_showEditor(e: Event) {
  const target = e.target;
  this.setState({edit: {
    row: parseInt(target.dataset.row, 10),
    key: target.dataset.key,
  }});
}
```

使用这种写法时，Flow 会提示错误：

```
js/source/components/Excel.js:87
 87:       row: parseInt(target.dataset.row, 10),
                                ^^^^^^^ property `dataset`. Property not found
in
 87:       row: parseInt(target.dataset.row, 10),
                         ^^^^^^ EventTarget
js/source/components/Excel.js:88
 88:       key: target.dataset.key,
                       ^^^^^^^ property `dataset`. Property not found in
 88:       key: target.dataset.key,
                ^^^^^^ EventTarget


Found 2 errors
```

如果你阅读 Flow 的源代码中关于 DOM 对象的定义（https://github.com/facebook/flow/blob/master/lib/dom.js），会发现 `EventTarget` 类型没有包含 `dataset` 属性，但 `HTMLElement` 类型却包含了该属性。因此进行类型转换来解决该问题：

```
const target = ((e.target: any): HTMLElement);
```

一开始你可能会觉得这个语法有点奇怪，但是可以把这个语句理解成通过圆括号包裹起来的值、冒号和类型。这个语法能让类型为 A 的值变成类型 B。在这个例子中，值本身没有发生变化，但是由 `any` 类型变为了 `HTMLElement` 类型。

### 7.3.9 invariant

在 `Excel` 组件中，使用两个状态属性跟踪用户是否正在编辑一个单元格以及对话框是否显示：

```
this.state = {
  // ...
```

```
  edit: null, // {row index, schema.id},
  dialog: null, // {type, idx}
};
```

这两个属性值可能是 `null`（没有进入编辑状态，对话框不显示），也可能是包含了编辑或对话框信息的对象。因此，这两个属性的类型如下：

```
type EditState = {
  row: number,
  key: string,
};

type DialogState = {
  idx: number,
  type: string,
};

type State = {
  data: Data,
  sortby: ?string,
  descending: boolean,
  edit: ?EditState,
  dialog: ?DialogState,
};
```

目前代码中大致存在的问题是，这些值有时候为 `null`，有时候不为 `null`。Flow 会对此有些疑问，事实上这也是有道理的。当你试图使用 `this.state.edit.row` 或者 `this.state.edit.key` 时，Flow 会提示错误：

```
Property cannot be accessed on possibly null value
```

虽然你知道只有当它们可用时才会调用它们，但 Flow 并不能判断这种情况。随着应用增长，你也难以保证以后不会忽略某些难以预料的状态。因此，当这种情况发生时，最好还是加以注意。要解决 Flow 的报错，并在应用出现问题时及时抛出异常，你可以通过以下方式检查代码中的所有非空值。

修改前：

```
data[this.state.edit.row][this.state.edit.key] = value;
```

修改后：

```
if (!this.state.edit) {
  throw new Error('Messed up edit state');
}
data[this.state.edit.row][this.state.edit.key] = value;
```

现在问题已经解决了，但这种条件判断的代码显得有点啰嗦。你可以使用一个 `invariant()` 函数替代这种写法。既可以选择自己实现该函数，也可以使用已有的开源代码包。

通过 NPM 安装 `invariant`：

```
$ npm install --save-dev invariant
```

然后在 .flowconfig 中添加配置：

```
[include]
node_modules/react
node_modules/react-dom
node_modules/classnames
node_modules/invariant
```

现在调用该函数：

```
invariant(this.state.edit, 'Messed up edit state');
data[this.state.edit.row][this.state.edit.key] = value;
```

# 7.4 测试

要确保稳健地改进应用，接下来需要关注自动化测试。提到测试这个话题，依然有很多开源项目可供选择。React 库使用了 Jest 工具（http://facebook.github.io/jest/）进行测试，因此我们选择介绍 Jest，看它对我们有什么帮助。此外，React 还提供了一个名为 `react-addons-test-utils` 的插件包，可以配合你进行测试工作。

首先进行安装配置。

## 7.4.1 安装

安装 Jest 的命令行界面：

```
$ npm i -g jest-cli
```

此外还需安装 `babel-jest`（让你可以使用 ES6 风格编写测试）以及 React 的测试工具包：

```
$ npm i --save-dev babel-jest react-addons-test-utils
```

接下来，更新 package.json：

```
{
  /* ... */
  "eslintConfig": {
    /* ... */
    "env": {
      "browser": true,
      "jest": true
    },
  /* ... */
  "scripts": {
```

```
    "watch": "watch \"sh scripts/build.sh\" js/source js/__tests__ css/",
    "test": "jest"
  },
  "jest": {
    "scriptPreprocessor": "node_modules/babel-jest",
    "unmockedModulePathPatterns": [
      "node_modules/react",
      "node_modules/react-dom",
      "node_modules/react-addons-test-utils",
      "node_modules/fbjs"
    ]
  }
}
```

现在你可以在命令行直接运行 Jest：

```
$ jest testname.js
```

也可以通过 `npm` 运行：

```
$ npm test testname.js
```

Jest 会在名为 `__tests__` 的文件夹中寻找测试用例，因此我们在 `js/__tests__` 目录中编写测试。

最后修改构建脚本，每次构建时都需要运行 lint 以及测试：

```
# QA
eslint js/source js/__tests__
flow
npm test
```

还要修改 watch.sh，以监听测试目录中的文件更改（别忘了我们在 package.json 文件中重复编写过这个功能）：

```
watch "sh scripts/build.sh" js/source js/__tests__ css/
```

## 7.4.2　首个测试

Jest 是基于流行的测试框架 Jasmine 构建的，而 Jasmine 的 API 命名比较口语化，简洁易懂。一开始，你需要通过 `describe('suite', callback)` 定义测试套件（test suite），在套件中通过 `it('test name', callback)` 定义一个或多个测试用例（test spec），并且在每一个用例中通过 `expect()` 函数进行断言（assertion）。

整体而言，其基础骨架大致如下：

```
describe('A suite', () => {
  it('is a spec', () => {
```

```
    expect(1).toBe(1);
  });
});
```

运行该测试：

```
$ npm test js/__tests__/dummy-test.js

> whinepad@2.0.0 test /Users/stoyanstefanov/reactbook/whinepad2
> jest "js/__tests__/dummy-test.js"

Using Jest CLI v0.8.2, jasmine1
 PASS js/__tests__/dummy-test.js (0.206s)
1 test passed (1 total in 1 test suite, run time 0.602s)
```

当你的测试中存在错误的断言时，比如：

```
expect(1).toBeFalsy();
```

测试程序会运行失败，并给出错误信息，如图 7-1 所示。

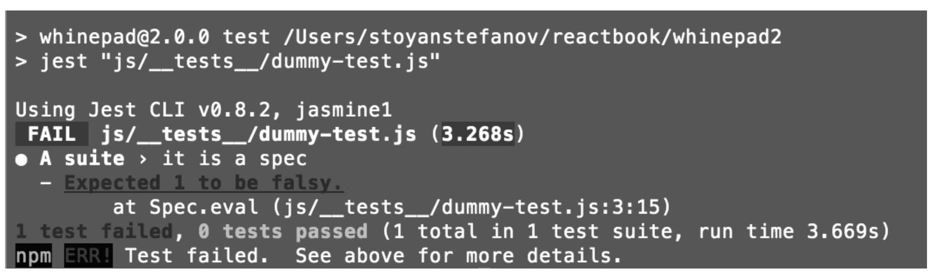



图 7-1：测试运行失败

### 7.4.3　首个 React 测试

接下来介绍 Jest 在 React 世界中的应用，你可以先从测试一个简单的 DOM 按钮开始。首先导入依赖：

```
import React from 'react';
import ReactDOM from 'react-dom';
import TestUtils from 'react-addons-test-utils';
```

设置测试套件：

```
describe('We can render a button', () => {
  it('changes the text after click', () => {
    // ...
  });
});
```

既然模板代码准备好了，接下来就要开始进行渲染和测试了。首先渲染一些简单的 JSX：

```
const button = TestUtils.renderIntoDocument(
  <button
    onClick={ev => ev.target.innerHTML = 'Bye'}>
    Hello
  </button>
);
```

在这里，我们使用了 React 的测试工具库渲染 JSX——当你点击按钮时，文本内容会发生改变。

在内容渲染完成后，需要检查渲染内容是否符合设想：

```
expect(ReactDOM.findDOMNode(button).textContent).toEqual('Hello');
```

如你所见，这里使用了 `ReactDOM.findDOMNode()` 获取 DOM 节点。随后，你可以使用熟悉的 DOM API 检查该节点。

你通常还需要测试界面与用户的交互。React 提供了 `TestUtils.Simulate` 对象，方便你完成这件事情：

```
TestUtils.Simulate.click(button);
```

最后需要检查界面是否响应了交互事件：

```
expect(ReactDOM.findDOMNode(button).textContent).toEqual('Bye');
```

本章的剩余部分将会介绍更多例子和 API，其中主要用到的工具如下：

- `TestUtils.renderIntoDocument`（用于渲染任意 JSX）；
- `TestUtils.Simulate.*` 负责与界面进行交互；
- `ReactDOM.findDOMNode()`（或者其他一些 `TestUtils` 方法）负责取得 DOM 节点的引用，然后检查节点内容是否符合设想。

### 7.4.4　测试 <Button> 组件

`<Button>` 组件的代码如下所示：

```
/* @flow */

import React from 'react';
import classNames from 'classnames';

type Props = {
  href: ?string,
  className: ?string,
};
```

```
const Button = (props: Props) =>
  props.href
    ? <a {...props} className={classNames('Button', props.className)} />
    : <button {...props} className={classNames('Button', props.className)} />

export default Button
```

我们将测试如下功能：

- 根据 href 属性是否存在，对应渲染 <a> 标签或 <button> 标签（第一个测试用例）；
- 接收自定义类名（第二个测试用例）。

创建一个新的测试文件：

```
jest
  .dontMock('../source/components/Button')
  .dontMock('classnames')
;

import React from 'react';
import ReactDOM from 'react-dom';
import TestUtils from 'react-addons-test-utils';
```

这里的 import 语句没有变化，但前面加上了新的 jest.dontMock() 调用。

这里 mock 的含义是指使用某些模拟的代码替代原有的功能，以便进行测试。mock 在单元测试中很常用，因为你希望测试的是一个“单元”——隔离测试某个模块，以减少整个系统中其他模块的影响。人们通常要花费相当多的努力在编写 mock 上，而 Jest 则采取了相反的做法：自动为每个依赖的模块生成 mock，并默认提供 mock。当你希望测试真实代码而不是 mock 的时候，可以通过 dontMock() 方法设置不需要进行 mock。[1]

在上述例子中，你声明了不希望 mock <Button> 组件及其使用的 classnames 库。

接下来引入 <Button> 组件：

```
const Button = require('../source/components/Button');
```

在本书编写时，尽管在 Jest 文档中也采用这种写法，但上述 require() 调用不能正常工作。你需要将其修改为：

```
const Button = require('../source/components/
Button').default;
```

注 1：从 Jest 15.0 版本开始，自动 mock 的功能已经被禁用了，具体说明参见 http://facebook.github.io/jest/blog/2016/09/01/jest-15.html#disabled-automocking。

——译者注

import 语句也不能生效：

```
import Button from '../source/components/Button';
```

需要修改为：

```
import _Button from '../source/components/Button';
const Button = _Button.default;
```

另一种做法是在 <Button> 组件中把 export default Button 修改为 export {Button}，然后通过 import {Button} from '../source/component/Button' 进行导入。

希望在读者阅读本书时，默认的 import 语句已经可以正常工作。

### 1. 第一个用例

接下来，我们分别使用 describe() 方法和 it() 方法设置测试套件和第一个测试用例：

```
describe('Render Button components', () => {
  it('renders <a> vs <button>', () => {
    /* 渲染组件并检测结果 */
  });
});
```

我们先渲染一个简单的按钮组件。它没有 href 属性，因此实际应该渲染一个 <button> 标签：

```
const button = TestUtils.renderIntoDocument(
  <div>
    <Button>
      Hello
    </Button>
  </div>
);
```

注意，<Button> 这种无状态函数式组件需要包裹在另一层 DOM 节点中，以便随后通过 ReactDOM 获取。

现在调用 ReactDOM.findDOMNode(button) 会返回包裹层 <div>。因此要获取 <button>，需要取得第一个子节点，并检查确认该节点是否为按钮：

```
expect(ReactDOM.findDOMNode(button).children[0].nodeName).toEqual('BUTTON');
```

类似地，你还需要在这个测试用例中验证当 href 属性存在时是否会渲染 <a> 标签。

```
const a = TestUtils.renderIntoDocument(
  <div>
    <Button href="#">
      Hello
    </Button>
  </div>
);
expect(ReactDOM.findDOMNode(a).children[0].nodeName).toEqual('A');
```

### 2. 第二个用例

在第二个测试用例中，你需要添加自定义类名，并检查最终生成的类名是否符合预期：

```
it('allows custom CSS classes', () => {
  const button = TestUtils.renderIntoDocument(
    <div><Button className="good bye">Hello</Button></div>
  );
  const buttonNode = ReactDOM.findDOMNode(button).children[0];
  expect(buttonNode.getAttribute('class')).toEqual('Button good bye');
});
```

这里有必要重点强调 Jest 的 mock 功能。有时，你可能在编写测试时发现没有得到预期的结果。这种情况有可能是因为你忘记关闭了 Jest 的 mock 功能。比如，你可以尝试把顶部的代码改为：

```
jest
  .dontMock('../source/components/Button')
  // .dontMock('classnames')
;
```

这时 Jest 会 mock 你的 `classnames` 模块，并使得该模块不会实现任何功能。你可以通过以下测试证明这个结果：

```
const button = TestUtils.renderIntoDocument(
  <div><Button className="good bye">Hello</Button></div>
);
console.log(ReactDOM.findDOMNode(button).outerHTML);
```

这段代码在控制台中生成的 HTML 代码如下：

```
<div data-reactid=".2">
  <button data-reactid=".2.0">Hello</button>
</div>
```

如你所见，无论填写什么类名，最终都不会生成出来，因为 `classNames()` 在 mock 之后不会实现任何功能。

把注释掉的 `dontMock()` 方法还原：

```
jest
  .dontMock('../source/components/Button')
  .dontMock('classnames')
;
```

然后 `outerHTML` 就会发生变化：

```
<div data-reactid=".2">
  <button class="Button good bye" data-reactid=".2.0">Hello</button>
</div>
```

这时你的测试就可以成功通过了。

> 当一个测试的执行不正常时，你可能需要知道实际生成的 HTML 结构是什么。一个快速简便的方法就是使用 `console.log(node.outerHTML)`，让 HTML 内容在控制台中输出。

## 7.4.5 测试 <Actions> 组件

`<Actions>` 组件也是一个无状态组件，这意味着你需要把它包裹在另一层 DOM 节点中，以便随后进行检查。一种方式是像前面对 `<Button>` 组件所做的那样，将其包裹在 `div` 中并通过以下方式访问：

```
const actions = TestUtils.renderIntoDocument(
  <div><Actions /></div>
);

ReactDOM.findDOMNode(actions).children[0]; // <Actions>组件的根节点
```

### 1. 组件包裹层

另一种方式是使用组件包裹层，以方便你随后使用 `TestUtils` 提供的各种方法寻找需要检查的节点。

这个包裹层非常简单，可以定义在一个模块中，方便以后重用：

```
import React from 'react';
class Wrap extends React.Component {
  render() {
    return <div>{this.props.children}</div>;
  }
}
export default Wrap
```

接下来编写测试模板：

```
jest
  .dontMock('../source/components/Actions')
  .dontMock('./Wrap')
;

import React from 'react';
import TestUtils from 'react-addons-test-utils';

const Actions = require('../source/components/Actions');
const Wrap = require('./Wrap');

describe('Click some actions', () => {
```

```
    it('calls you back', () => {
      /* 渲染 */
      const actions = TestUtils.renderIntoDocument(
        <Wrap><Actions /></Wrap>
      );
      /* 搜索并检查节点 */
    });
  });
```

### 2. mock 函数

<Actions> 组件并没有什么特别的地方。我们回忆一下，其代码如下：

```
const Actions = (props: Props) =>
  <div className="Actions">
    <span
      tabIndex="0"
      className="ActionsInfo"
      title="More info"
      onClick={props.onAction.bind(null, 'info')}>&#8505;</span>
    {/* 另外两个span标签 */}
  </div>
```

唯一需要测试的功能是，当点击这些按钮时，能否正确地调用 onAction 回调函数。Jest 允许你定义 mock 函数，并验证函数如何被调用。这用于验证我们的回调函数再合适不过了。

在测试代码中，你需要创建一个新的 mock 函数，并将其以回调函数的形式传递给 Actions：

```
const callback = jest.genMockFunction();
const actions = TestUtils.renderIntoDocument(
  <Wrap><Actions onAction={callback} /></Wrap>
);
```

接下来模拟点击动作按钮：

```
TestUtils
  .scryRenderedDOMComponentsWithTag(actions, 'span')
  .forEach(span => TestUtils.Simulate.click(span));
```

注意我们使用了 TestUtils 中的一个方法寻找 DOM 节点。该方法返回一个包含三个 <span> 节点的数组，然后你逐一点击数组中的每个节点。

现在你的 mock 回调函数必然会被调用三次。你需要在 expect() 方法中对此进行断言：

```
const calls = callback.mock.calls;
expect(calls.length).toEqual(3);
```

如你所见，callback.mock.calls 属性是一个数组。每当函数被调用时，都会传递一个包含参数的数组到该属性中。

第一个动作按钮的名称是 info，在回调时通过 props.onAction.bind(null, 'info') 的形式

把类型 info 传递到回调函数中。因此，第一次调用 mock 回调（0）的第一个参数（0）必然是 info：

```
expect(calls[0][0]).toEqual('info');
```

另外两个按钮的断言也类似：

```
expect(calls[1][0]).toEqual('edit');
expect(calls[2][0]).toEqual('delete');
```

3. find 与 scry

TestUtils（https://facebook.github.io/react/docs/test-utils.html）提供了一系列函数，帮助你在 React 渲染树中寻找 DOM 节点。比如，根据标签名或者类名寻找节点。前面的例子用到了其中一种方法：

```
TestUtils.scryRenderedDOMComponentsWithTag(actions, 'span')
```

另一种方法是：

```
TestUtils.scryRenderedDOMComponentsWithClass(actions, 'ActionsInfo')
```

与 scry* 方法相对应的是 find* 方法。比如：

```
TestUtils.findRenderedDOMComponentWithClass(actions, 'ActionsInfo')
```

注意上述方法中 Component 和 Components 的区别。scry* 系列方法找出所有匹配的节点，并返回一个数组（甚至在只匹配一个或者零个节点时也是如此），而 find* 系列方法只返回一个节点。如果没有匹配节点或者匹配了多个节点时，后者就会报错。因此，当你使用 find* 系列方法进行寻找时，已经意味着假定在 DOM 树中只存在一个 DOM 节点。

## 7.4.6　更多模拟交互

接下来测试 Rating 组件。在鼠标移入、移出和点击时，其状态会发生改变。测试的模板代码如下：

```
jest
  .dontMock('../source/components/Rating')
  .dontMock('classnames')
;

import React from 'react';
import TestUtils from 'react-addons-test-utils';

const Rating = require('../source/components/Rating');

describe('works', () => {
  it('handles user actions', () => {
```

```
    const input = TestUtils.renderIntoDocument(<Rating />);

    /* 在此编写expect()的期望结果 */

  });
});
```

注意这里不需要在渲染时包裹 `<Rating>` 组件，因为这个组件不是无状态函数式组件。

组件中有很多星星（默认为 5 个），每个 `span` 标签对应一颗星星。我们可以通过以下方式获取它们：

```
const stars = TestUtils.scryRenderedDOMComponentsWithTag(input, 'span');
```

现在测试需要模拟鼠标移入和移出事件，并点击第 4 颗星星（`span[3]`）。这时，前面 4 颗星星都应该被“点亮”，也就是包含 `RatingOn` 类名，而第 5 颗星星应该维持原有的“熄灭”状态：

```
TestUtils.Simulate.mouseOver(stars[3]);
expect(stars[0].className).toBe('RatingOn');
expect(stars[3].className).toBe('RatingOn');
expect(stars[4].className).toBeFalsy();
expect(input.state.rating).toBe(0);
expect(input.state.tmpRating).toBe(4);

TestUtils.Simulate.mouseOut(stars[3]);
expect(stars[0].className).toBeFalsy();
expect(stars[3].className).toBeFalsy();
expect(stars[4].className).toBeFalsy();
expect(input.state.rating).toBe(0);
expect(input.state.tmpRating).toBe(0);

TestUtils.Simulate.click(stars[3]);
expect(input.getValue()).toBe(4);
expect(stars[0].className).toBe('RatingOn');
expect(stars[3].className).toBe('RatingOn');
expect(stars[4].className).toBeFalsy();
expect(input.state.rating).toBe(4);
expect(input.state.tmpRating).toBe(4);
```

此外还要注意测试是如何获取组件状态的，以验证 `state.rating` 和 `state.tmpRating` 的正确性。对于测试而言，这可能有点苛刻了。毕竟如果外部的显示结果符合预期，为何还要关心组件内部的状态呢？此处只是为了证明这样做是可行的。

### 7.4.7　测试完整的交互

接下来为 `Excel` 组件编写测试。毕竟这个组件是整个应用的关键，一旦该组件出现问题，就会严重影响应用体验。事不宜迟，开始编写测试：

```
jest.autoMockOff();

import React from 'react';
import TestUtils from 'react-addons-test-utils';

const Excel = require('../source/components/Excel');
const schema = require('../source/schema');

let data = [{}];
schema.forEach(item => data[0][item.id] = item.sample);

describe('Editing data', () => {
  it('saves new data', () => {
    /* 渲染组件、模拟交互、检查节点 */
  });
});
```

首先要注意到顶部的 `jest.autoMockOff();` 函数。这里没有逐一列出 `Excel` 使用的所有组件（以及组件内使用的组件），你可以通过这个方法直接禁用所有的 mock。

接下来，需要用 `schema` 对象和示例数据 `data` 对组件进行初始化工作（和 app.js 类似）。

然后进行渲染：

```
const callback = jest.genMockFunction();
const table = TestUtils.renderIntoDocument(
  <Excel
    schema={schema}
    initialData={data}
    onDataChange={callback} />
);
```

目前看起来一切正常。现在改变第一行的第一个单元格。设置新的值为：

```
const newname = '$2.99 chuck';
```

目标单元格为：

```
const cell = TestUtils.scryRenderedDOMComponentsWithTag(table, 'td')[0];
```

在本书编写时，需要编写一些额外的 hack 代码以支持访问 `dataset` 属性，因为 Jest 尚未支持这种 DOM 操作：

```
cell.dataset = { // 针对 Jest 的 DOM 兼容性问题采取的非常规手段
  row: cell.getAttribute('data-row'),
  key: cell.getAttribute('data-key'),
};
```

双击单元格，其内容会变为一个包含文本输入框的表单：

```
TestUtils.Simulate.doubleClick(cell);
```

改变输入框的值，并提交表单：

```
cell.getElementsByTagName('input')[0].value = newname;
TestUtils.Simulate.submit(cell.getElementsByTagName('form')[0]);
```

现在单元格的内容不再是表单了，而是纯文本内容：

```
expect(cell.textContent).toBe(newname);
```

此时 `onDataChange` 回调函数会被调用。该函数接收一个数组参数，数组中包含了表格数据的键值对。你可以验证 mock 回调函数是否正确接收到了新的数据：

```
expect(callback.mock.calls[0][0][0].name).toBe(newname);
```

此处 `[0][0][0]` 的含义为：第一个 0 对应 mock 函数第一次被调用；第二个 0 对应函数接收的首个参数，它在这里是数组；第三个 0 对应数组中的第一个元素，它在这里是一个对象（对应表格中的一条记录），其中的 `name` 属性值为 $2.99 chuck。

> 除了使用 `TestUtils.Simulate.submit` 提交表单，你还可以选用 `TestUtils.Simulate.keyDown` 模拟敲下回车键时的事件。这同样可以提交表单。

现在编写第二个测试用例，我们删除一行示例数据：

```
it('deletes data', () => {
  // 和之前相同
  const callback = jest.genMockFunction();
  const table = TestUtils.renderIntoDocument(
    <Excel
      schema={schema}
      initialData={data}
      onDataChange={callback} />
  );

  TestUtils.Simulate.click( // 点击图标
    TestUtils.findRenderedDOMComponentWithClass(table, 'ActionsDelete')
  );

  TestUtils.Simulate.click( // 确认对话框
    TestUtils.findRenderedDOMComponentWithClass(table, 'Button')
  );

  expect(callback.mock.calls[0][0].length).toBe(0);
});
```

在前面的例子中，`callback.mock.calls[0][0]` 对应用户交互后产生的新数据。但这个例子中的数组是空的，因为在测试中删除了单行记录。

### 7.4.8 代码覆盖率

掌握了上述这些主题后，剩下的事情就简单了，但可能会有一点枯燥。你需要确保尽可能多地测试所有可能发生的场景。比如点击 info 行为按钮并点击取消，点击删除并点击取消，再次点击并删除。

测试是很有必要的，可以帮助你更快地解决问题、更自信地进行开发与代码重构。测试还可以帮助你纠正同事的错误：当他们觉得改变某一处地方不会造成什么影响时，事实可能远远超出他们的预想。一种“游戏化”测试过程的方式是使用*代码覆盖率*（code coverage）特性。

运行：

```
$ jest --coverage
```

该命令会运行找到的所有测试并生成一份报告，告诉你已经测试（或*覆盖*）了多少行代码、多少个函数等。示例结果如图 7-2 所示。

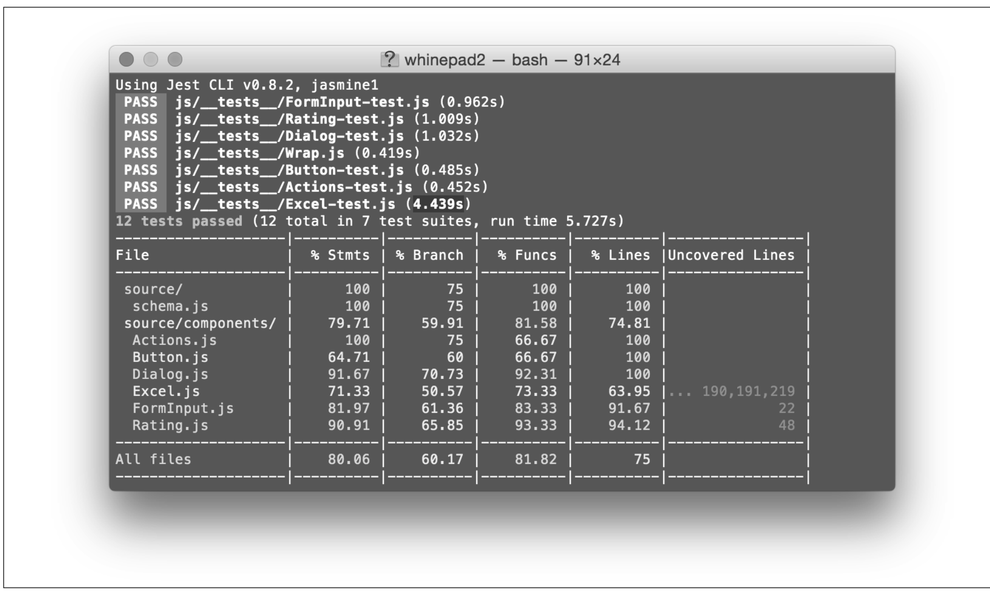



图 7-2：代码覆盖率报告

你会发现目前的测试报告并非完美，必然要编写更多的测试用例。代码覆盖率报告的一个实用特性是显示未被覆盖的行号。比如，尽管你已经对 `FormInput` 进行了测试，但第 22 行

还没有被覆盖。我们找出这行代码，发现它是一个 `return` 语句：

```
getValue(): FormInputFieldValue {
  return 'value' in this.refs.input
    ? this.refs.input.value
    : this.refs.input.getValue();
}
```

看来我们还没有测试过这个函数。因此赶紧编写一个测试用例作为补救措施：

```
it('returns input value', () => {
  let input = TestUtils.renderIntoDocument(<FormInput type="year" />);
  expect(input.getValue()).toBe(String(new Date().getFullYear()));
  input = TestUtils.renderIntoDocument(
    <FormInput type="rating" defaultValue="3" />
  );
  expect(input.getValue()).toBe(3);
});
```

第一个 `expect()` 测试了一个内建的 DOM 输入元素，而第二个 `expect()` 则测试了自定义的 `input` 组件。现在 `getValue()` 方法中的三元表达式对应的两种情况应该都会被覆盖了。

再次运行上述命令。在目前的代码覆盖率报告中，关于第 22 行的提示已经消失了（如图 7-3 所示）。

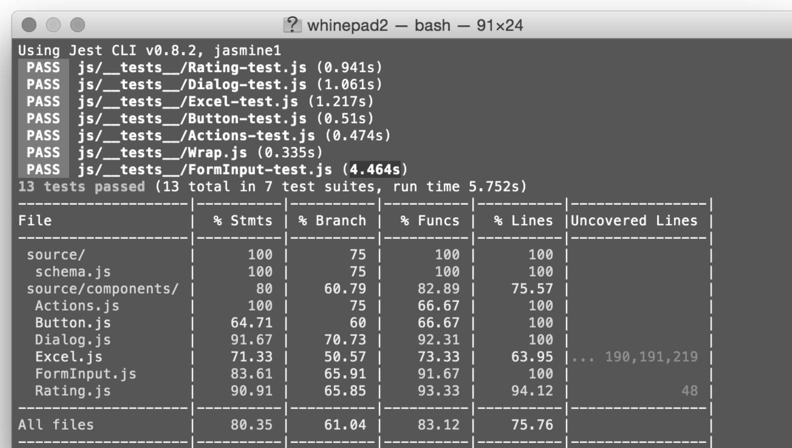



图 7-3：修改后的代码覆盖率报告

第 8 章

# Flux

最后一章将介绍 Flux（https://facebook.github.io/flux/）。Flux 是管理组件间通信的另外一种方式，同时也可以用于管理整个应用的数据流。目前我们已经知道如何把属性从父组件传递到子组件，从而进行组件间通信并监听子组件发生的变化（比如通过 `onDataChange`）。然而通过这种方式传递属性时，你可能会遇到一个组件需要接收很多属性的情况。这就使得组件的测试变得难以进行，因为属性的组合排列情况非常多，难以验证所有的情况能否正常工作。

此外，在某些场景中，你还需要把属性像“管道”般从父组件传递到子组件、第三级组件、第四级组件……然而这种工作往往是重复的（本质上是不良实践）、混乱的，并且会让阅读代码的人的精神负担更大（有太多事情需要同时跟踪）。

Flux 就是一种可以帮助你克服这些障碍的方式，并让你在管理应用的数据流动时保持头脑清晰。与其说 Flux 是一个代码库，不如说它是一种组织（架构）应用数据的思想。毕竟在大部分情况下，数据都是非常重要的。用户可能会使用你的应用管理他们的钱、邮件、照片等。如果应用界面有点粗糙，用户也许还能忍受；但应用在任何情况下都应该能正常处理数据，不能让用户产生疑惑（“刚才我到底有没有成功支付 30 美元？”）。

目前，基于 Flux 的思想已经衍生出许多版本的开源实现，但本章不会逐一讨论这些实现，而是将尝试自己实现一个 Flux 架构。一旦你掌握其思想（并确信 Flux 能给你带来便利），就可以深入探究这些开源实现并从中选取一种适合实际开发的方案，也可以持续完善自己的方案。

# 8.1 理念

在 Flux 的理念中，应用的核心就是数据。数据存储于 Store 中。你的 React 组件（View，视图）从 Store 中读取数据并进行渲染。用户与应用之间的交互行为会产生 Action（比如点击按钮）。Action 会导致 Store 中的数据发生变化，进而影响 View。这个循环一直持续进行（如图 8-1 所示）。在循环中，数据流动是单向的（unidirectional）。这种单向数据流的优点是更容易进行跟踪、推理与调试。

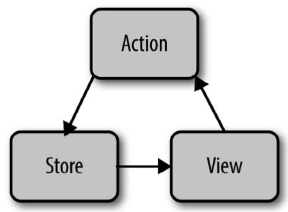



图 8-1：单向数据流

在上述架构的基础之上，还有一些变种与延伸版本，包括更多 Action、多重 Store 以及 Dispatcher 等。在深入解释这些新概念之前，我们先阅读一部分代码。

# 8.2 回顾 Whinepad

在 Whinepad 应用中，顶层的 React 组件称为 <Whinepad>，其创建方式如下：

```
<Whinepad
  schema={schema}
  initialData={data} />
```

<Whinepad> 组件负责构成 <Excel> 组件：

```
<Excel
  schema={this.props.schema}
  initialData={this.state.data}
  onDataChange={this._onExcelDataChange.bind(this)} />
```

首先，描述应用所需数据的 schema 对象从 <Whinepad> 组件（像管道般）传递到 <Excel> 组件中（接下来再传递到 <Form> 组件）。这种方式显得有些重复和纵向模式化。假如你还需要管道式地传递几个类似的属性呢？这样下去，组件*表面*（surface）就会变得过于庞大，且毫无益处。

> 上文提及的“表面”指的是一个组件接收的所有属性，通常被用作“API”与“函数签名”的同义词。在程序设计中，应当保持表面最小化。一个函数若接收多达十个参数，会比接收两个（或零个）参数更难以使用、调试与测试。

虽然 schema 对象是按照原样传递的，但是数据看似并非如此。<Whinepad> 组件接收到 initialData 属性后，会对数据进行某些修改再传递给 <Excel> 组件（传递的属性是 this.state.data 而不是 this.props.initialData）。这里会产生两个问题：如果新的数据不同于原有数据，会发生什么？当提及最新的数据时，到底哪个组件拥有“单一数据源”？

在上一章的实现中，虽然顶层 <Whinepad> 组件确实包含了最新的数据，但这并没有明确为什么 UI 组件（React 是完全和 UI 相关的）应该掌握数据的来源。

接下来介绍如何把这项工作移交给 Store 进行处理。

## 8.3 Store

首先复制一份现有的代码：

```
$ cd ~/reactbook
$ cp -r whinepad2 whinepad3
$ cd whinepad3
$ npm run watch
```

接下来，创建一个新目录用于放置 Flux 模块（以便与 React 的 UI 组件区分开）。目前只需创建 Store 和 Actions 两个模块：

```
$ mkdir js/source/flux
$ touch js/source/flux/CRUDStore.js
$ touch js/source/flux/CRUDActions.js
```

Flux 架构允许你创建多个 Store（比如一个关于用户数据，另一个关于应用设置等），但目前我们只关注单个 CRUD Store 的情况。这个 Store 全权负责处理一个记录列表；在这个例子中，是关于酒类及其评价的记录。

这个 CRUDStore 和 React 本身没有联系，只需使用一个简单的 JavaScript 对象即可实现：

```
/* @flow */

let data;
let schema;

const CRUDStore = {

  getData(): Array<Object> {
    return data;
  },

  getSchema(): Array<Object> {
    return schema;
  },
```

```
};

export default CRUDStore
```

如你所见，这个 Store 负责维护单一的数据来源，比如本地模块变量 `data` 和 `schema`，并且可以把这些变量返回给任何需要的地方。此外，Store 还允许更新 `data` 变量（但不更新 `scheam` 变量，因为这是一个贯穿整个应用的常量）：

```
setData(newData: Array<Object>, commit: boolean = true) {
  data = newData;
  if (commit && 'localStorage' in window) {
    localStorage.setItem('data', JSON.stringify(newData));
  }
  emitter.emit('change');
},
```

在上述代码中，除了更新本地的 `data` 变量以外，Store 还更新了持久化存储中的数据；在这个例子中数据存储在 `localStorage`，而在实际情况中还有可能是向服务器发起 XHR 请求。这通常只会在“提交”情况下发生，因为没有必要总是更新持久化存储。比如在搜索时，你总是想要取得最新的数据，因此没有必要永久地保存这些数据。但是假如在调用 `setData()` 后停电，你丢失了除搜索结果外的所有数据呢？

最后，你会看到一个 `change` 事件被触发。（稍后你将了解更多内容。）

Store 还可以提供一些有用的方法，比如获取数据的总行数，以及获取某一行记录：

```
getCount(): number {
  return data.length;
},

getRecord(recordId: number): ?Object {
  return recordId in data ? data[recordId] : null;
},
```

在应用初始化时，你需要初始化 Store。在此之前初始化工作通过 app.js 进行，但现在 Store 是处理数据的唯一地方，因此将数据初始化交给 Store 自行处理：

```
init(initialSchema: Array<Object>) {
  schema = initialSchema;
  const storage = 'localStorage' in window
    ? localStorage.getItem('data')
    : null;

  if (!storage) {
    data = [{}];
    schema.forEach(item => data[0][item.id] = item.sample);
  } else {
    data = JSON.parse(storage);
  }
},
```

现在 app.js 可以通过以下方式启动应用：

```
// ...
import CRUDStore from './flux/CRUDStore';
import Whinepad from './components/Whinepad';
import schema from './schema';

CRUDStore.init(schema);

ReactDOM.render(
  <div>
    {/* 更多JSX代码 */}
    <Whinepad />
  {/* ... */}
);
```

如你所见，一旦 Store 被初始化，`<Whinepad>` 组件就不需要再接收任何属性了。组件所需的数据可以通过 `CRUDStore.getData()` 方法获取，而数据的描述可以通过 `CRUDStore.getSchema()` 获取。

> 你可能会问：为何 Store 不能自己读取数据，而是依赖于外部传入的 `schema` 对象？其实，你当然可以直接在 Store 中导入 `schema` 模块，但让应用自身处理 `schema` 的来源更为合理。试想 `schema` 是否为一个模块？若直接导入是否属于硬编码？既然是模块，是否应该由用户定义？

## 8.3.1 Store 事件

还记得之前在 Store 更新数据时调用的 `emitter.emit('change');` 方法吗？这个方法用于通知对数据感兴趣的 UI 模块，以便模块在数据发生变化时从 Store 中读取最新的数据，进行自我更新。那么，这个事件触发机制到底是如何实现的呢？

实现事件订阅的模式有很多。从本质上说，这些模式需要收集数据的一系列相关者（订阅者），然后在事件发生时“推送”消息，调用每个订阅者的回调函数（订阅者在订阅事件时，需要提供这个回调函数）。

为简单起见，我们使用一个名为 `fbemitter` 的小型开源库来实现事件订阅功能：

```
$ npm i --save-dev fbemitter
```

更新 .flowconfig 文件：

```
[ignore]
.*/fbemitter/node_modules/.*
# 省略部分代码

[include]
```

```
node_modules/classnames
node_modules/fbemitter
# 省略部分代码
```

在 Store 模块的开端，需要导入并初始化事件 emitter：

```
/* @flow */

import {EventEmitter} from 'fbemitter';

let data;
let schema;
const emitter = new EventEmitter();

const CRUDStore = {
  // ...
};

export default CRUDStore
```

这个 emitter 需要负责两项任务：

- 收集订阅；
- 通知订阅者（正如之前看到的那样，在 setData() 方法中调用 emitter.emit('change')）。

在 Store 中，还可以把收集订阅功能作为一个方法暴露出来，让调用者无需了解其实现细节：

```
const CRUDStore = {
  // ...
  addListener(eventType: string, fn: Function) {
    emitter.addListener(eventType, fn);
  },
  // ...
};
```

至此，这个 CRUDStore 的功能已经完备了。

## 8.3.2 在 <Whinepad> 中使用 Store

借助 Flux 实现 <Whinepad> 组件相当简单，因为其中的大部分功能将迁移到 CRUDActions 中实现（稍后介绍），但 CRUDStore 也发挥了不少作用。我们将不再需要维护 this.state.data。之前需要维护它的原因仅仅是需要把状态传递给 <Excel> 组件，但现在 <Excel> 组件可以通过 Store 取得数据了。事实上，<Whinepad> 组件甚至不需要处理 Store。我们不妨添加一个需要用到 Store 的功能：在搜索域中显示记录总数（如图 8-2 所示）。

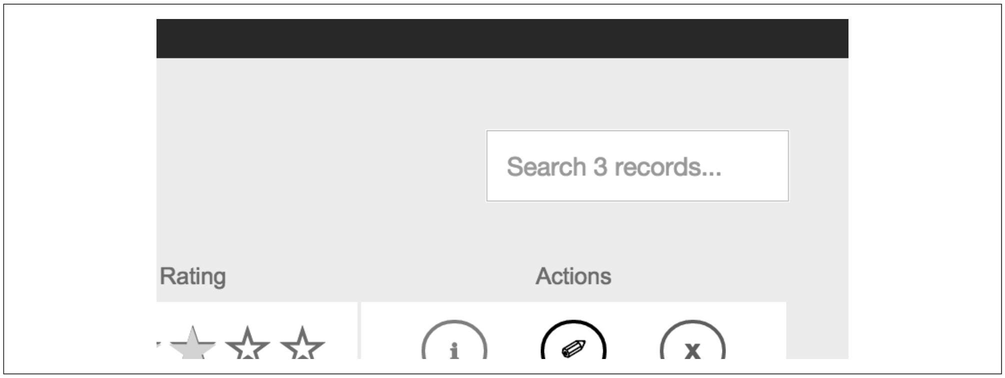



图 8-2：在搜索区域中显示记录总数

之前，在 <Whinepad> 组件的 constructor() 构造函数中，需要对 state 进行初始化：

```
this.state = {
  data: props.initialData,
  addnew: false,
};
```

现在不再需要 data 属性了，但你仍需一个 count 变量记录数据总计数。因此在初始化时，需要从 Store 中读取数据：

```
/* @flow */

// ...
import CRUDStore from '../flux/CRUDStore';
// ...

class Whinepad extends Component {
  constructor() {
    super();
    this.state = {
      addnew: false,
      count: CRUDStore.getCount(),
    };
  }

  /* ... */
}

export default Whinepad
```

此外，在构造函数中还需要订阅 Store 的数据变化事件，以便在数据变化时有机会更新 this.state 中的数据总计数：

```
constructor() {
  super();
  this.state = {
```

```
    addnew: false,
    count: CRUDStore.getCount(),
  };

  CRUDStore.addListener('change', () => {
    this.setState({
      count: CRUDStore.getCount(),
    })
  });
}
```

以上就是组件和 Store 之间需要进行的所有交互。无论 Store 中的数据在任何时候、通过任何方式发生变化（包括调用 `CRUDStore` 中的 `setData()` 方法时），Store 都会触发一个 `change` 事件。`<Whinepad>` 组件会监听这个 `change` 事件并更新自身状态。你已经知道，设置状态将会导致组件重新渲染，因此 `render()` 方法会再次被调用。该方法中的逻辑和以往一样，仅仅是基于 `state` 和 `props` 组合 UI：

```
render() {
  return (
    {/* ... */}
    <input
      placeholder={this.state.count === 1
        ? 'Search 1 record...'
        : `Search ${this.state.count} records...`
      }
    />
    {/* ... */}
  );
}
```

此外，还可以通过在 `<Whinepad>` 组件中实现 `shouldComponentUpdate()` 方法来优化性能。由于数据的改变不一定会影响数据的总条数（比如编辑一条记录或者编辑记录中的某个字段），组件在这种情况下不需要重新渲染：

```
shouldComponentUpdate(newProps: Object, newState: State): boolean {
  return (
    newState.addnew !== this.state.addnew ||
    newState.count !== this.state.count
  );
}
```

最后，`<Whinepad>` 组件不再需要把 `data` 和 `schema` 传递到 `<Excel>` 组件中了。同样也不需要设置 `onDataChange` 回调函数，因为现在所有的数据改变都可以从 Store 的 `change` 事件中获知。因此，`<Whinepad>` 组件中的 `render()` 方法只需要这样写：

```
render() {
  return (
    {/* ... */}
    <div className="WhinepadDatagrid">
```

```
      <Excel />
    </div>
    {/* ... */}
  );
}
```

### 8.3.3 在 <Excel> 中使用 Store

和 `<Whinepad>` 组件类似，`<Excel>` 组件也不再需要接收属性了。构造函数从 Store 中读取 `schema`，并赋值给 `this.schema`。事实上，`this.state.schema` 和 `this.schema` 之间并没有实质区别，只不过 `state` 意味着变量可能会发生某些变化，而 `schema` 是一个常量。

至于数据，现在只需要在初始化时从 Store 中读取并记录在 `this.state.data` 中即可，不再通过属性接收。

最后，构造函数订阅了 Store 的 `change` 事件，以便 state 中的数据保持最新（并触发重新渲染）：

```
constructor() {
  super();
  this.state = {
    data: CRUDStore.getData(),
    sortby: null, // schema.id
    descending: false,
    edit: null, // {row index, schema.id},
    dialog: null, // {type, idx}
  };
  this.schema = CRUDStore.getSchema();
  CRUDStore.addListener('change', () => {
    this.setState({
      data: CRUDStore.getData(),
    })
  });
}
```

得益于 Store，`<Excel>` 组件需要做的就是这么简单。`render()` 方法和之前一样，只需从 `this.state` 中读取并呈现数据即可。

也许你会感到疑惑：为何需要把 Store 中的数据复制到 `this.state` 中？是否可以在 `render()` 方法中直接访问 Store 中的数据呢？虽然理论上是可行的，但这样做会使组件会变得“不纯”。还记得 2.15 节提到过*纯渲染组件*（pure render component）的概念吗？这意味着组件渲染的内容仅由 `props` 和 `state` 决定。在 `render()` 方法中，任何函数调用看起来都是不可靠的——你不能确定在一个额外的函数调用中会返回什么内容。应用还会变得难以调试、难以预测：“为什么 state 中的数据是 1，而渲染出的内容是 2 呢？哦，原来是在 `render()` 方法中进行了函数调用。”

## 8.3.4　在 `<Form>` 中使用 Store

在此之前，表单组件同样需要获取 `schema`（用于获取 `fields` 属性）以及 `defaultValues` 属性（用于预填充表单或者显示一份只读版本）。目前这两者都已经转移到了 Store 中。因此现在表单只需要接收一个 `recordId` 属性，并根据该属性在 Store 中查找具体数据：

```
/* @flow */

import CRUDStore from '../flux/CRUDStore';

// ...

type Props = {
  readonly?: boolean,
  recordId: ?number,
};

class Form extends Component {
  fields: Array<Object>;
  initialData: ?Object;

  constructor(props: Props) {
    super(props);
    this.fields = CRUDStore.getSchema();
    if ('recordId' in this.props) {
      this.initialData = CRUDStore.getRecord(this.props.recordId);
    }
  }

  // ...
}

export default Form
```

表单组件不需要注册 Store 的 `change` 事件，因为在编辑表单时不需要监听数据的变化。但是可能出现这样的场景：另一个用户在同时进行编辑；同一个用户在不同的标签页中打开了这个应用，并在两边都编辑了数据。如果需要处理这些情况，你就需要监听数据的变化，并提醒用户：数据在其他地方正在被编辑。

## 8.3.5　界定

该如何界定是使用 Flux Store 还是使用借助属性的非 Flux 实现呢？ Store 为所有的数据需求提供了一站式服务，使你从属性传递中解脱出来，但与此同时也降低了组件的可重用性。现在，你不能在另一个完全不同的场景中直接重用 `Excel` 组件了，因为在组件中从 `CRUDStore` 获取数据的逻辑已经被硬编码。即便如此，只要新的场景中使用了类似 CRUD 的逻辑（这是有可能的，否则你为何需要可编辑的数据表格呢？），你还是可以让组件从 Store 中获取数据。谨记一点：在应用中可以根据需要使用任意数量的 Store。

对于那些底层组件，比如按钮和表单输入元素，最好不要使用 Store。因为对于这些组件来说，使用传递属性的方式更加方便。那些处于两个极端间的组件类型都属于灰色地带［从最简单的按钮（比如 `<Button>`）到最顶层负责管理所有内容的父组件（比如 `<Whinepad>`）］，由你来界定是否使用 Flux。`<Form>` 组件是应该像之前那样连接到 CRUD Store 还是应该和 Store 隔离使其可重用呢？你可以根据手头上的任务以及将来重用该组件的可能性，作出最合适的选择。

## 8.4 Action

Action 描述了 Store 中的数据如何发生改变。当用户和 View 交互时，用户执行的操作（即 Action）会改变 Store，并会把该事件传递到受影响的 View 中。

要实现这个更新 `CRUDStore` 的 `CRUDActions` 很简单，只需要使用另一个常规的 JavaScript 对象：

```
/* @flow */

import CRUDStore from './CRUDStore';

const CRUDActions = {
  /* 具体方法 */
};

export default CRUDActions
```

### 8.4.1 CRUD Action

在 `CRUDActions` 模块中，我们应该实现哪些类型的方法呢？最常见的猜测是 `create()`、`delete()`、`update()` 等。在这个应用中，唯一特别的地方是我们可以选择更新整条记录或者更新某个特定的表单域。因此，需要分别实现 `updateRecord()` 和 `updateField()` 两种方法：

```
/* @flow */
/* ... */
const CRUDActions = {

  create(newRecord: Object) {
    let data = CRUDStore.getData();
    data.unshift(newRecord);
    CRUDStore.setData(data);
  },

  delete(recordId: number) {
    let data = CRUDStore.getData();
    data.splice(recordId, 1);
    CRUDStore.setData(data);
  },
```

```
  updateRecord(recordId: number, newRecord: Object) {
    let data = CRUDStore.getData();
    data[recordId] = newRecord;
    CRUDStore.setData(data);
  },

  updateField(recordId: number, key: string, value: string|number) {
    let data = CRUDStore.getData();
    data[recordId][key] = value;
    CRUDStore.setData(data);
  },

  /* ... */
};
```

这些代码看起来相当琐碎：你需要从 Store 中读取当前数据，然后对其进行处理，最后把新的数据写入 Store。

> 在这里不需要实现 CRUD 中的 R，因为 Store 已经为你提供了读取数据的功能。

## 8.4.2 搜索与排序

在之前的实现中，`<Whinepad>` 组件负责搜索数据。这样做的原因仅仅是因为搜索域刚好在这个组件的 `render()` 方法当中。事实上，这个功能应该放在更靠近数据的地方。

类似地，排序功能在之前属于 `<Excel>` 组件，因为排序的组件放置在表格头部，并且表头的 `onClick` 事件处理器负责进行排序。不过同样，排序功能最好也放置在与数据相关的地方。

那么数据搜索与排序的功能应该放置在 Action 还是 Store 中呢？这可能会引起一些争议，因为这两种实现看起来都可行。在我们的实现中，我们让 Store“木偶化”，只负责进行 `get`、`set` 以及发送事件。Action 则是负责加工数据的地方，因此我们把排序与搜索功能从 UI 组件中抽离出来，并放到 `CRUDActions` 模块中：

```
/* @flow */
/* ... */
const CRUDActions = {

  /* CRUD方法 */

  _preSearchData: null,
```

```
  startSearching() {
    this._preSearchData = CRUDStore.getData();
  },

  search(e: Event) {
    const target = ((e.target: any): HTMLInputElement);
    const needle: string = target.value.toLowerCase();
    if (!needle) {
      CRUDStore.setData(this._preSearchData);
      return;
    }
    const fields = CRUDStore.getSchema().map(item => item.id);
    if (!this._preSearchData) {
      return;
    }
    const searchdata = this._preSearchData.filter(row => {
      for (let f = 0; f < fields.length; f++) {
        if (row[fields[f]].toString().toLowerCase().indexOf(needle) > -1) {
          return true;
        }
      }
      return false;
    });
    CRUDStore.setData(searchdata, /* commit */ false);
  },

  _sortCallback(
    a: (string|number), b: (string|number), descending: boolean
  ): number {
    let res: number = 0;
    if (typeof a === 'number' && typeof b === 'number') {
      res = a - b;
    } else {
      res = String(a).localeCompare(String(b));
    }
    return descending ? -1 * res : res;
  },

  sort(key: string, descending: boolean) {
    CRUDStore.setData(CRUDStore.getData().sort(
      (a, b) => this._sortCallback(a[key], b[key], descending)
    ));
  },

};
```

至此，CRUDActions 的功能已经完备了。接下来看看 <Whinepad> 和 <Excel> 组件是如何使用这个 CRUDActions 的。

你或许会认为，上述的 search() 函数不应该放在 CRUDActions 中：

```
search(e: Event) {
  const target = ((e.target: any): HTMLInputElement);
  const needle: string = target.value.toLowerCase();
  /* ... */
}
```

或许 Action 模块不应该关心界面的内容，因此更合理的处理方法应该是：

```
search(needle: string) {
  /* ... */
}
```

这个参数看起来更加合理，你也确实可以采用这种方法。然而，这种做法会对 <Whinepad> 组件造成一些影响，因为你不能直接使用 <input onChange="CRUDActions.search">，而是需要进行一些额外的处理。

### 8.4.3 在 <Whinepad> 中使用 Action

在 Flux Action 创建完成后，来看看 <Whinepad> 组件目前的版本。首先要做的当然是引入 Action 模块：

```
/* @flow */

/* ... */
import CRUDActions from '../flux/CRUDActions';
/* ... */

class Whinepad extends Component {/* ... */}

export default Whinepad
```

也许你还记得，Whinepad 组件是负责添加新记录以及搜索现有记录的（如图 8-3 所示）。




图 8-3：Whinepad 组件负责处理的数据区域

在过去，当谈及添加新记录的功能时，Whinepad 组件需要负责操作自身的 this.state.data：

```
_addNew(action: string) {
  if (action === 'dismiss') {
    this.setState({addnew: false});
  } else {
    let data = Array.from(this.state.data);
    data.unshift(this.refs.form.getData());
    this.setState({
      addnew: false,
      data: data,
    });
    this._commitToStorage(data);
  }
}
```

不过目前由 Action 模块负责更新 Store（称作单一数据源），减轻了 Whinepad 组件的负担：

```
_addNew(action: string) {
  this.setState({addnew: false});
  if (action === 'confirm') {
    CRUDActions.create(this.refs.form.getData());
  }
}
```

现在不需要维护更多的状态，也不需要操作其他数据了。如果产生了用户操作，只需要分发这个 Action，让其顺着单向的数据流动即可。

与之类似，搜索功能在过去也是通过组件自身的 this.state.data 完成的，而现在只需：

```
<input
  placeholder={this.state.count === 1
    ? 'Search 1 record...'
    : `Search ${this.state.count} records...`
  }
  onChange={CRUDActions.search.bind(CRUDActions)}
  onFocus={CRUDActions.startSearching.bind(CRUDActions)} />
```

### 8.4.4　在 <Excel> 中使用 Action

Excel 组件需要使用 CRUDActions 模块提供的排序、删除、修改功能。我们之前的删除方法如下：

```
_deleteConfirmationClick(action: string) {
  if (action === 'dismiss') {
    this._closeDialog();
    return;
  }
  const index = this.state.dialog ? this.state.dialog.idx : null;
  invariant(typeof index === 'number', 'Unexpected dialog state');
  let data = Array.from(this.state.data);
  data.splice(index, 1);
  this.setState({
```

```
      dialog: null,
      data: data,
    });
    this._fireDataChange(data);
  }
```

现在把方法修改为：

```
  _deleteConfirmationClick(action: string) {
    this.setState({dialog: null});
    if (action === 'dismiss') {
      return;
    }
    const index = this.state.dialog && this.state.dialog.idx;
    invariant(typeof index === 'number', 'Unexpected dialog state');
    CRUDActions.delete(index);
  }
```

现在不需要手动触发数据改变的事件了，因为其他组件不再需要监听 `Excel` 组件，只需订阅 Store 中的数据改变事件即可。此外也不再需要操作 `this.state.data` 了，因为 Action 模块会代替组件进行这些操作，然后在 Store 触发事件时更新组件。

排序与修改记录的功能与之类似。所有的数据操作都只需要修改为调用 `CRUDActions` 中的某一个方法：

```
  /* @flow */

  /* ... */
  import CRUDActions from '../flux-imm/CRUDActions';
  /* ... */

  class Excel extends Component {

    /* ... */

    _sort(key: string) {
      const descending = this.state.sortby === key && !this.state.descending;
      CRUDActions.sort(key, descending);
      this.setState({
        sortby: key,
        descending: descending,
      });
    }

    _save(e: Event) {
      e.preventDefault();
      invariant(this.state.edit, 'Messed up edit state');
      CRUDActions.updateField(
        this.state.edit.row,
        this.state.edit.key,
        this.refs.input.getValue()
      );
```

```
    this.setState({
      edit: null,
    });
  }

  _saveDataDialog(action: string) {
    this.setState({dialog: null});
    if (action === 'dismiss') {
      return;
    }
    const index = this.state.dialog && this.state.dialog.idx;
    invariant(typeof index === 'number', 'Unexpected dialog state');
    CRUDActions.updateRecord(index, this.refs.form.getData());
  }

  /* ... */
};

export default Excel
```

Whinepad 应用迁移到 Flux 架构的完整版本可以从本书附带的代码库（https://github.com/stoyan/reactbook/）中下载。

## 8.5 Flux 回顾

目前为止，我们已经把应用迁移到了 Flux 架构中（尽管还显得有些粗糙）。你让 View 发送 Action，然后 Action 负责更新单一 Store 并发送事件。View 需要监听对应的事件并负责进行更新。这就形成了一个完整的循环。

这种思想还有一些延伸，能帮助你更好地应对应用规模增长。

Action 不仅局限于由 View 发送（如图 8-4 所示），还可以来源于服务端。比如以下这些情况：某些数据过时，需要更新；其他用户改变了某些数据，而应用需要从服务器同步数据；随着时间推移，需要采取一些操作（买票后没有在一定时间内付款，会话过期后需要重新购买）。

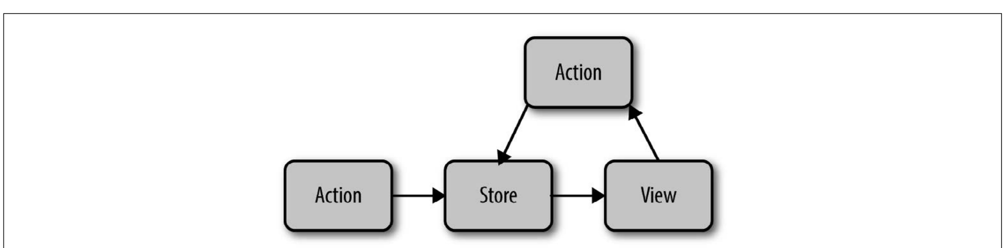



图 8-4：更多的 Action

当你碰到这种需要处理不同来源的 Action 时，单一 Dispatcher 的概念就变得很有帮助了（如图 8-5 所示）。Dispatcher 负责管理所有 Action，并将其传递到单个 Store 中（也可能是多个 Store）。

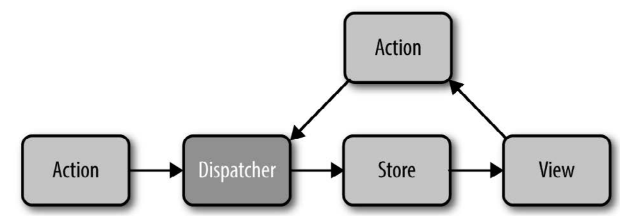



图 8-5：Dispatcher

在一些功能更丰富的应用中，你还会遇到更多不同的 Action。它们可能来源于 UI、服务器或者某些别的地方，因此需要多个 Store 负责处理各自对应的数据（如图 8-6 所示）。

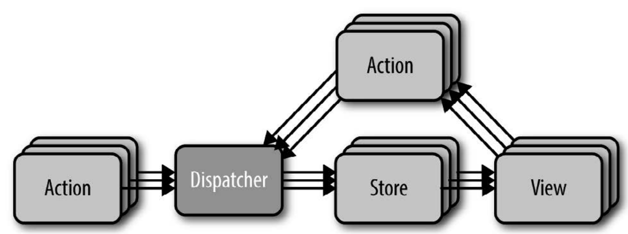



图 8-6：复杂的单向流

Flux 架构的实现还有许多开源的版本。在一开始，你可以先从简单的版本入手。随着应用规模增长，你可以选择继续改进自己实现的版本或者选择一种开源的解决方案。

## 8.6 immutable

在本书的最后，我们介绍一个关于 Flux 中 Store 和 Action 的修改方案，把酒类的数据记录切换到不可变的数据结构。在 React 应用中，immutable 是一个常见的主题，尽管它和 React 本身并没有什么关联。

immutable 对象在创建后是不能更改的。在通常情况下，immutable 对象比普通对象更加容易理解与推理。比如，字符串类型背后的原理就是通过 immutable 对象实现的。

在 JavaScript 中，可以通过 `npm` 安装 `immutable` 包实现这个功能：

```
$ npm i --save-dev immutable
```

更新 .flowconfig 文件：

```
# ....
[include]

# ...
node_modules/immutable

# ...
```

`immutable` 库提供了在线文档，参见 http://facebook.github.io/immutable-js/。

因为所有的数据处理都发生在 Store 和 Action 中，所以只有这两处需要更新。

### 8.6.1 immutable 存储数据

`immutable` 库提供了 `List`、`Stack` 和 `Map` 等数据类型。我们首先介绍 `List`，因为这和我们之前在应用中使用的数组类型相似：

```
/* @flow */

import {EventEmitter} from 'fbemitter';
import {List} from 'immutable';

let data: List<Object>;
let schema;
const emitter = new EventEmitter();
```

现在 `data` 变量被定义为 immutable 中的 `List` 类型。

可以通过 `let list = List()` 创建新的列表，并传递一些初始值。我们看看 Store 现在是如何初始化列表的：

```
const CRUDStore = {

  init(initialSchema: Array<Object>) {
    schema = initialSchema;
    const storage = 'localStorage' in window
      ? localStorage.getItem('data')
      : null;
    if (!storage) {
      let initialRecord = {};
      schema.forEach(item => initialRecord[item.id] = item.sample);
      data = List([initialRecord]);
    } else {
```

```
      data = List(JSON.parse(storage));
    }
  },

  /* ... */
};
```

如你所见，列表使用了一个数组进行初始化。初始化完成后，就可以使用列表的 API 进行数据操作。此外，一旦列表完成创建，这个列表就是不可变的，不能被修改。（但所有的数据操作都发生在 CRUDActions 中，随后你将会看到具体代码。）

在 Store 中，除了修改数据初始化和类型注解以外，不需要进行太多其他的修改——因为 Store 要做的全部工作仅仅关于 set 和 get。

需要在 getCount() 方法中进行一点修改，因为 immutable 中的列表没有 length 属性：

```
// 修改前
getCount(): number {
  return data.length;
},

// 修改后
getCount(): number {
  return data.count(); // 也可以使用data.size获取
},
```

最后需要修改 getRecord() 方法，因为 immutable 库不能像内建的数据类型那样通过数组下标进行取值：

```
// 修改前
getRecord(recordId: number): ?Object {
  return recordId in data ? data[recordId] : null;
},

// 修改后
getRecord(recordId: number): ?Object {
  return data.get(recordId);
},
```

## 8.6.2　immutable 数据操作

首先回顾一下 JavaScript 中的字符串方法：

```
let hi = 'Hello';
let ho = hi.toLowerCase();
hi; // "Hello"
ho; // "hello"
```

赋值给 hi 变量的字符串不会发生变化。当需要修改字符串时，会创建一个新的字符串。

immutable 中的列表类型也是这样：

```
let list = List([1, 2]);
let newlist = list.push(3, 4);
list.size; // 2
newlist.size; // 4
list.toArray(); // Array [ 1, 2 ]
newlist.toArray() // Array [ 1, 2, 3, 4 ]
```

注意到 `push()` 方法了吗？对于 immutable 中的列表，其大部分方法和原生数组类型中的方法类似，比如 `map()` 和 `forEach()` 等方法依然是可用的。这也是应用的 UI 逻辑不需要进行修改的真正原因。（准确地说，只需要修改方括号访问数组元素的地方。）另一个原因如前文所述：数组处理主要在 Store 和 Action 中发生。

事实上，数据结构的变化对于 Action 模块的影响并不大。因为 immutable 中的列表还提供了 `sort()` 和 `filter()` 方法，所以排序与搜索方法不需要修改。需要修改的只有 `create()`、`delete()` 和两个 `update*()` 方法。

思考如下 `delete()` 方法：

```
/* @flow */

import CRUDStore from './CRUDStore';
import {List} from 'immutable';

const CRUDActions = {

  /* ... */

  delete(recordId: number) {
    // 修改前：
    // let data = CRUDStore.getData();
    // data.splice(recordId, 1);
    // CRUDStore.setData(data);

    // 修改后：
    let data: List<Object> = CRUDStore.getData();
    CRUDStore.setData(data.remove(recordId));
  },

  /* ... */

};

export default CRUDActions;
```

JavaScript 中的 `splice()` 方法名字有点古怪。这个方法既会返回原数组中的某一部分，又会修改原有的数组。把这两种操作写在同一行中，难免会引起混淆。不过，immutable 列

表中的方法可以使用一行写法且不会引起混淆。如果我们不进行类型注解的话，还可以将其简化为：

```
delete(recordId: number) {
  CRUDStore.setData(CRUDStore.getData().remove(recordId));
},
```

在 immutable 世界中，`remove()` 方法的命名非常合理。这个方法不会影响原有的列表，因为原列表是不可变的。`remove()` 方法会返回一个新列表，并移除原列表中的某个元素。然后你可以把这个新列表保存到 Store 中。

其他的数据处理方法也与之类似，比直接操作数组更为简单：

```
/* ... */
create(newRecord: Object) { // 和数组类似,使用unshift()方法
  CRUDStore.setData(CRUDStore.getData().unshift(newRecord));
},

updateRecord(recordId: number, newRecord: Object) { // 使用set()方法代替[]
  CRUDStore.setData(CRUDStore.getData().set(recordId, newRecord));
},

updateField(recordId: number, key: string, value: string|number) {
  let record = CRUDStore.getData().get(recordId);
  record[key] = value;
  CRUDStore.setData(CRUDStore.getData().set(recordId, record));
},
/* ... */
```

大功告成！现在，你的应用中使用了下列技术栈：

- React 组件，用于定义 UI；
- JSX，用于组合组件；
- Flux，用于组织数据流；
- 不可变数据；
- Babel，帮助我们使用最新的 ECMAScript 特性；
- Flow，用于进行类型检查和语法检查；
- ESLint，用于检查更多的错误与代码约定；
- Jest，用于进行单元测试。

和之前一样，你可以在本书附带的代码库（https://github.com/stoyan/reactbook/）中获取这个 Whinepad 应用的第三个版本（使用了 immutable）。还可以在 http://whinepad.com 在线体验这个应用。

## 关于作者

Stoyan Stefanov 是 Facebook 的开发工程师，曾供职于 Yahoo。Stoyan 是图像优化工具 smush.it 的作者，同时也是性能优化工具 YSlow 2.0 的架构师。Stoyan 曾经编写了《JavaScript 模式》和《JavaScript 面向对象编程指南》，还为《高性能网站建设进阶指南》和《高性能 JavaScript》贡献过内容。Stoyan 的个人站点是 http://phpied.com。他还在许多会议中发表过演讲，包括 Velocity、JSConf 和 Fronteers 等。

## 关于封面

本书封面上的动物是镰嘴管舌雀。作者的女儿在完成了一份关于这种动物的作业报告后，选择它作为本书的封面动物。镰嘴管舌雀在夏威夷岛屿最常见的本土鸟类中排名第三，但是其家族中的许多物种已经灭绝或濒临灭绝。这种颜色鲜艳的小型鸟类主要生活在夏威夷岛、茂宜岛和可爱岛，是夏威夷地区的标志性象征。

成年的镰嘴管舌雀多数是鲜红色的，带有黑色的翅膀和尾巴，喙弯曲。由于鲜红色与周围的绿色叶子形成了鲜明的对比，镰嘴管舌雀在野外非常容易辨认。尽管其羽毛被广泛用于装饰夏威夷贵族的斗篷和头盔，但镰嘴管舌雀还没有到濒临灭绝的地步。这是因为当地人认为它与其近亲夏威夷监督吸蜜鸟相比显得没那么神圣。

镰嘴管舌雀主要以花蜜和多型铁心木为食，偶尔也吃一些小昆虫。镰嘴管舌雀还会进行垂直迁徙，一年四季都会跟随开花的时间进行海拔高度上的迁徙。这意味着镰嘴管舌雀会在岛屿之间迁徙，但因为生态环境的破坏，镰嘴管舌雀在瓦胡岛和莫洛凯岛非常罕见，而且自 1929 年就已经在拉奈岛绝迹了。

为了保护现存的镰嘴管舌雀种群，人们已经采取了一些举措。这种鸟类非常容易感染鸟痘和禽流感，此外还会遭受砍伐森林和外来入侵植物的影响。因为野猪挖的泥坑会滋生蚊虫，所以封锁森林区域有助于控制蚊子传播疾病。目前正在进行的工作还包括恢复森林和清除外来植物物种，这都有利于镰嘴管舌雀喜欢吸食的花种繁衍。O'Reilly 封面上的许多动物都已经濒临灭绝，然而每一种物种对于地球都非常重要。要想知道如何为此贡献你的一份力量，请到 animals.oreilly.com 进一步了解。

封面图片来自 Wood 的 *Illustrated Natural History* 一书。

O'REILLY®

# React快速上手开发

本书旨在帮你掌握Facebook的开源技术React，迅速建立富Web应用，构建组件并将其组织成可维护的大型应用程序。

解开Web应用开发之谜，从了解React基本原理开始。

- 设置React并编写第一个Hello World应用
- 创建并使用自定义React组件以及通用DOM组件
- 构建一个可以编辑、排序、搜索和导出内容的数据表格组件
- 使用JSX语法扩展作为调用函数的替代选择
- 设置一个帮你集中注意力于React上的简单构建过程
- 构建一个可以将数据存储在客户端的完整自定义应用
- 在应用规模增长时使用ESLint、Flow和Jest等工具检查并测试代码
- 使用Flux管理组件间的通信

“这本书可以真正帮你为构建自己的React应用奠定坚实的基础。”

——Andreea Manole
Facebook开发工程师

**Stoyan Stefanov**，Facebook开发工程师，图像优化工具smush.it的作者，性能优化工具YSlow2.0的架构师。曾多次在Velocity等技术大会上发表过演讲。另著有《JavaScript模式》和《JavaScript面向对象编程指南》，还为《高性能网站建设进阶指南》和《高性能JavaScript》贡献过内容。个人站点是http://phpied.com。

PROGRAMMING/JAVASCRIPT

封面设计：Randy Comer　张健

图灵社区：iTuring.cn
热线：(010)51095186转600

分类建议　计算机／程序设计/Web开发

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版



ISBN 978-7-115-44773-9
定价：49.00元